Executing and Verifying Higher-Order Functional-Imperative Programs in Maude

Vlad Rusu[a], Andrei Arusoaie[b]

[a] *Inria Lille Nord Europe, France* vlad.rusu@inria.fr
[b] *"Alexandru Ioan Cuza" University of Iasi, Romania* andrei.arusoaie@uaic.ro

Abstract

We incorporate higher-order functions and state monads in Maude, thereby embedding a higher-order functional language with imperative features in the Maude framework. We illustrate, via simple programs in the resulting language: the concrete and symbolic execution of programs; their verification with respect to properties expressed in Reachability Logic, a language-parametric generalisation of Hoare Logic; and the verification of program-equivalence properties. Our approach is proved sound and is implemented in Full Maude by taking advantage of its reflective features and module system.

Keywords: Maude, higher-order function, state monad, Reachability Logic.

1. Introduction

Maude [1] is a high-performance logical and semantical framework based on equational and rewriting logic. At its core, Maude is not a higher-order language, in the sense that it does not have functional types; thus, functions cannot be passed as arguments or returned as values of other functions. This limitation in expressiveness is, however, largely made up for by other features of the language: the *module system* for parameterised programming, and *reflectiveness* for manipulating Maude constructions as meta-level terms.

Contribution. In this paper we use the module system and the meta-level in order to incorporate higher-order functions and state monads in Maude. This amounts to a shallow embedding in Maude of a higher-order functional programming language with imperative features. In the resulting language one can program in the spirit of, e.g., Haskell [2], while still having access to the familiar (for Maude users) features of the Maude language and system.

We illustrate the resulting language with simple imperative-functional programs. We show how programs can be: executed with concrete and symbolic data; formally verified with respect to properties expressed in Reachability Logic [3, 4, 5, 6], a language-independent generalisation of Hoare logic; and formally proved equivalent. The soundness of symbolic execution, program verification and program-equivalence verification is formally proved.

Related Work. Higher-order functions are common in most functional languages. In Maude, the possibility to define higher-order functions is known[1] but this feature (including the definition and evaluation of lambda-functions) has not been, to our best knowledge, fully exploited before. Perhaps one reason is that in order to implement higher-order functional constructs one needs advanced features currently not available in (core) Maude but only in *Full Maude*, a Maude extension also implemented in Maude by reflection.

Monads are an essential ingredient in some functional languages such as Haskell. In particular, state monads are used to introduce imperative features without compromising the purity of a functional programming language. State monads have also been introduced in the Coq proof assistant [7] in order to prove programs written in an imperative style [8]. Our integration of state monads in Maude is to our best knowledge new in this framework.

Another related line of work is program verification, specifically, with respect to Reachability-Logic (RL) formulas. RL is both a formalism for defining programming language semantics and a program-specification logic that can be seen as a language-parametric generalisation of Hoare logics. In order to verify programs in a language, say, $\mathcal{L}$, one defines the operational semantics of $\mathcal{L}$, then specifies expected properties of the program $\mathcal{P}$ of interest; the program correctness is defined as a semantic consequence between the RL formulas defining the semantics of $\mathcal{L}$ and those defining the properties of $\mathcal{P}$. This approach [9] has been used on languages/programs defined in the $\mathbb{K}$ framework [10]. One contribution of this paper is showing that the previous approach can also be used on languages defined by *shallow embedding*, whereas existing works use *deep embeddings*. An advantage of a shallow embedding is that one can use constructions of the host language when writing programs in guest language. By contrast, in a deep embedding programs have to stay within the bounds of the guest language's syntax and

[1] see, e.g., the answer to a question regarding this on the Maude help list at `https://lists.cs.illinois.edu/lists/arc/maude-help/2006-01/msg00005.html`)

semantics. The same dichotomy (shallow vs. deep embeddings) distinguishes the present paper with our previous work on language-parametric symbolic execution, program verification and program equivalence [11, 12, 13, 14, 15].

There are of course many other works in these areas, which, in contrast to ours, are mostly dedicated to specific languages. Symbolic execution - running programs with symbolic values instead of concrete ones - has been introduced in the seventies [16] and has been used for debugging, testing and analysing programs. Tools including symbolic execution in combination with other ingredients include Java PathFinder [17], DART [18], CUTE [19], EXE [20], PEX [21]. Tools that build upon symbolic execution to perform program analysis and verification include [22, 23, 24, 25]. Regarding program equivalence, one can mention works on C compiler correctness [26, 27], and approaches targetting specific classes of languages: functional [28], microcode [29], CLP [30]. Some approaches target particular kinds of programs: successive versions of a given piece of code [31], recursive procedures [32].

In the general area of rewriting, model-checking techniques based on narrowing have been developped [33]. The most significant difference with our approach is that narrowing-based techniques only allow unconditional rewrite rules, which we do allow (and intensively use) in this paper; on the other hand, they deal with linear-temporal logic, which is more expressive than reachability logic. Last but not least, *rewriting modulo* SMT [34, 35] is a theory and implementation of symbolic execution in Maude, essentially isomorphic to the one we developed for $\mathbb{K}$ [36] - an example of simultaneous discovery and materialisation of concepts that are "in the air" at a certain moment in time.

Finally we compare the present paper with the workshop version [37]. The first difference is that we here apply symbolic execution, program verification and program equivalence to a shallow-embedded language in Maude, with higher-order functional-imperative features, whereas in the earlier version we applied formal verification only, to a program in a simple imperative language deeply embedded in Maude. Another difference is that in the early version we insisted on *incrementality* in the proof, whereas here this feature is not dominant (yet remains nonetheless present). The third and last difference is the inclusion of all proofs in this paper in order to make it self-contained.

The Maude source code for the work reported in this paper is available at: `https://profs.info.uaic.ro/~arusoaie.andrei/maude-monad.zip`.

```
fth TRIV is
    sort Elt . *** this is a comment
endfth
```

Figure 1: Theory `TRIV`.

Organisation. After this introduction in Section 2 we introduce our approach for incorporating higher-order functions and state monads into Maude. In Section 3 we present Reachability Logic (RL) and symbolic execution based on languages whose operational semantics is defined using RL. The relations between concrete and symbolic executions are stated. In Section 4 we introduce a tree-construction procedure based on symbolic execution, together with a result saying that, if the procedure terminates successfully on a set of RL formulas specifying a program, then the program does satisfy the formulas. In Section 5 we show how program verification can help in establishing program equivalence. All concepts are illustrated on simple higher-order functional-imperative programs. Conclusions and future work directions are drawn in Section 6. An Appendix contains detailed proofs for all the results.

2. Higher-Order Functional-Imperative Programs in Maude

In this section we introduce Maude and show how higher-order functions and state monads can be incorporated in it. We illustrate the approach with simple programs, which can be seen as belonging to a higher-order functional-imperative language shallowly embedded in Maude. The Maude language is introduced gradually via the concepts presented throughout this section.

2.1. Adding Higher-Order Functions to Maude

Maude is an executable specification language. Maude specifications (which we shall sometimes call *programs*) are structured into units called *modules* and *theories*, which can import each other, can be parameterised, and can be instantiated by linking actual parameters to formal parameters of a parameterised unit. There exist a number of predefined units for common data structures (Booleans, integers, strings,...) collected into a Maude file called the *prelude*. Possibly the simplest unit from the prelude is the theory `TRIV`, which just declares one *sort* `Elt`. This theory is shown in Figure 1.

Figure 2 shows an example of a *functional module*, one of the several kinds of modules available in Maude. It has two formal parameters, `X,Y` that

```
fmod ARROW{X :: TRIV, Y :: TRIV} is *** formal parameters
protecting SUBST . *** imported module
sort Arrow{X,Y} .  *** sort declaration
op __ : Arrow{X,Y} X$Elt -> Y$Elt . *** operation declarations
op lambda_:_ : X$Elt Y$Elt -> Arrow{X,Y} [ctor] .
var f :  Arrow{X,Y} . *** variable declarations
vars x z : X$Elt .
var y : Y$Elt .
op ERR : -> [Y$Elt] . *** constant declaration
eq (lambda x : y)(z) = *** equation
   downTerm(subst(upTerm(x),upTerm(z),upTerm(y)), ERR)  .
endfm
```

Figure 2: Module `ARROW`.

are instances of theory `TRIV`. (The difference between modules and theories is not relevant here). This module imports another module `SUBST` that we will be discussing below. The `protecting` keyword indicates that the importation is not meant to modify the imported module's semantics. Then, a parameterised sort `Arrow{X,Y}` is declared, which, in our intention, is the sort of functions from `X` to `Y`. More precisely, since `X,Y` are not sorts but formal parameters for the predefined theory `TRIV`, the parameterised sort `Arrow{X,Y}` corresponds to the functions from `X$Elt` and `Y$Elt` where `Elt` is the sort defined in parameter `TRIV`.

Of course, just declaring the sort `Arrow{X,Y}` does not make it the sort of functions from `X$Elt` and `Y$Elt`: more information is needed. The declaration `op lambda_:_ : X$Elt Y$Elt -> Arrow{X,Y} [ctor]` is a part of this information. It says that elements in the sort `Arrow{X,Y}` can be *constructed* (`[ctor]`) with the *operation* `lambda_:_` that we use to denote anonymous functions. The operation takes an `X$Elt` parameter that is intended to be the function's argument, and a `Y$Elt` that is intended to be the function's body. Thus, terms of the form `lambda x : y`, with `x, y` declared in the module as *variables* of appropriate sorts, have the sort `Arrow{X,Y}`.

We can now syntactically construct lambda functions. Their semantics (by evaluation) is declared in the module as the *juxtaposition* operation `__ : Arrow{X,Y} X$Elt -> Y$Elt`, that is, the juxtaposition of a function (of sort `Arrow{X,Y}`) and an argument (of sort `X$Elt`) that, in our intention,

produces the result (of sort `Y$Elt`) obtained by "applying" the function on the argument. Thus, we can construct terms `(lambda x : y)(z)` denoting the application of the function `(lambda x : y)` to the argument `z`.

Unsurprisingly, the evaluation is achieved by substituting the formal parameter `x` with the actual parameter `z` in the function's body `y`. We achieve this using the imported module `SUBST` where an adequate operation `subst()` is defined. In order to work for terms of any sort (and not just the sorts we have declared in our module) the substitution operation is defined on the *metarepresentation* of `x`, `y`, and `z`. This is achieved by *raising* them to terms of a predefined sort `Term` available in a predefined module `META-LEVEL`, using the operation `upTerm()` also defined in that module. The substitution also produces a result of sort `Term`, which then has to be *lowered* back to (hopefully) a term of sort `Y$Elt`. This is achieved by the predefined operation `downTerm()` in `META-LEVEL`, which is given a term of sort `Term` and attempts to lower it to a given sort. The sort in question has to be specified by the user (as the Maude system has no way of knowing her intentions). Hence the declaration `ERR : -> [Y$Elt]` of a constant `ERR` of the *kind* `[Y$Elt]` - kinds are "supersets" of sorts that allow users to declare "error" terms in addition to the "proper" terms in the sorts. Thus, in our case, `downTerm()` attempts to convert the term `subst(upTerm(x),upTerm(z),upTerm(y))` of sort `Term` to a term of sort `Y$Elt`, and if it does not succeed it returns the constant `ERR`. This is the overall meaning of the last statement in our module, the *equation* `(lambda x : y)(z)=downTerm(subst(upTerm(x),upTerm(z),upTerm(y)),ERR)`.

Thus, most of the actual work is performed in the module `SUBST`, which we only briefly describe here. The module imports `META-LEVEL` in order to have access to the sort `Term` and to its *subsorts* `Variable` and `Constant`. The substitution operation (of a variable by a term in a term) is standard, except for situation when the `lambda_:_` operation is involved: it does not substitute a variable that is bound by `lambda_:_`, and in the case where the argument of the `lambda_:_` occurs in the term to be substituted the argument is first renamed before substitution is applied. This is expressed by the equations in Figure 3, the second of which is *conditional*. Expressions such as `'lambda_:_[x,t']` in the figure are Maude's meta-representations (of sort `Term`) for terms `lambda x : t'` (of sort `Y$Elt`).

We note that the current implementation does allow users to write nonsensical lambda-terms such as `lambda 1 : 1`. However, when one attempts to eveluate such terms on arguments, the result is the predefined constant `ERR`. This is quite similar to what occurs in general programming languages: many programs that do not make sense are accepted by parsers; they only generate errors at runtime.

We chose to define the `lambda` operation in this way because the alternative - expressing everything at Maude's metalevel - would make the code of higher-order

```
fmod SUBST is
protecting META-LEVEL .
...
eq subst(x,t, 'lambda_:_[x,t']) = 'lambda_:_[x,t'] .
ceq subst(x,t, 'lambda_:_[x',t']) =
   'lambda_:_[rename(x'),subst(x,t,subst(x', rename(x'), t'))]
if occurs(x',t)  .
...
endfm
```

Figure 3: Module SUBST (excerpt).

```
view Int from TRIV to INT is
   sort Elt to Int
endv
```

Figure 4: View Int.

functions unreadable, and even more so for state-monadic constructions built on top of higher-order functions.

Instantiating the ARROW module. We now instantiate the ARROW module in order to create functions from, say, Int to Int, where the predefined sort Int is defined in the Maude prelude in the module INT. Intuitively, one has to "link" the formal parameters X and Y to the module INT. In Maude this is done by a construction called a *view*. Since X and Y are parameters for the theory TRIV in Figure 1, the following (predefined) view connects the sort Elt of TRIV to Int of INT (Figure 4).

The name of the view is arbitrary, but choosing it to be the name of the sort that one wants as actual parameter (the one to which Elt is mapped in the view) allows us to build the module ARROW{Int,Int}, which gives the "illusion" that the formal parameters X, Y are (here) mapped to the sort Int. The module ARROW{Int,Int} is just like ARROW{X,Y} in Figure 2 except that it declares the sort ARROW{Int,Int} and the sorts XElt, YElt are now both Int. After loading the module ARROW{Int,Int} in Maude, let us ask Maude to evaluate the term lambda x:Int : x:Int + 1 and then the term (lambda x:Int : x:Int + 1) 3:

```
Maude> (reduce in ARROW{Int,Int} : (lambda x:Int : x:Int + 1) .)
result Arrow{Int,Int} :
lambda x:Int : x:Int + 1
```

```
Maude> (reduce in ARROW{Int,Int} : (lambda x:Int : x:Int + 1) 3 .)
result NzNat :
4
```

That is, we have asked Maude to perform *equational reduction* for the above terms. The sorts of the expected results, as well as the results themselves, are correct.

Functions of Several Arguments. What we have so far is a type for lambda-functions of one argument. The type for functions of several arguments exploits the idea of *currying*, which in our case amounts to having a sort `Arrow{X,Arrow{Y,Z}}` for functions of type `X` × `Y` to `Z`. Thus, we need a module `ARROW{X,Arrow{Y,Z}}`.

Remember, however that formal module parameters are linked to actual parameters via *views*, and that by convention the views have the same name as the type they want to pass as actual parameter. Hence, we need a *parameterised* view:

```
(view Arrow{X :: TRIV, Y :: TRIV} from TRIV to ARROW{X,Y} is
    sort Elt to Arrow{X,Y} .
endv)
```

It turns out that such parameterised views are not available in (Core) Maude, but they are available in Full Maude, a reflective implementation of Maude in Maude with some useful extensions (and a few minor differences). Hence hereafter we will be working in Full Maude. The most visible feature of Full Maude is that code units and commands are enclosed into parentheses, as were the last few ones shown above. Full Maude can also load Core Maude code units hence, which are then written without parentheses as were the ones shown earlier in Figures 1-3.

This concludes the presentation of higher-order functions in Maude. To illustrate it further, here is a command demonstrating a simple *partial evaluation*:

```
(reduce in ARROW{Int,Arrow{Int,Int}} :
            (lambda x:Int : (lambda y:Int : x:Int + y:Int)) 1 .)
result Arrow{Int,Int} :
  lambda y:Int : y:Int + 1
```

The result is the expected one (modulo the commutativity of addition in Maude).

2.2. State Monads

A state monad with *state* sort `S` and *output* sort `A` is a function from `S` to pairs (`A`, `S`). Assume a parameterised module `PAIR{X::TRIV,Y::TRIV}` defining a sort `Pair{X,Y}` with constructor `(_,_)` and accessors `fst` and `snd` linked by the appropriate equations. Then we define the parameterised sort `Monad{A,S}` of monads with state sort `S` and output sort `A` to be the sort `Arrow{S,Pair{A,S}}`:

```
(fmod MONAD {A :: TRIV, S :: TRIV} is
protecting ARROW{S,Pair{A,S}}
           * (sort  Arrow{S,Pair{A,S}} to Monad{A,S}) .
endfm)
```

The *renaming* construction `*` is here used to rename the sort `Arrow{S,Pair{A,S}}` `ARROW{S,Pair{A,S}}` to the more informative and concise name `Monad{A,S}`.

Together with monads two operations are usually defined : `ret`, which "returns" a given value of the output type without modifying the state, and `bind`, which "binds" the result of a computation to an identifier so that the value can be reused in subsequent computations. We define them in two separate modules.

```
(fmod RET {A :: TRIV, S :: TRIV} is
protecting ARROW{A, Monad{A,S}} .
var a : A$Elt .
var s : S$Elt .
 op ret : -> Arrow{A,Monad{A,S}} .
 eq ret(a)(s)  = (a,s)  .
endfm)
```

In the above module, `ret` is defined to be a constant of sort `Arrow{A,Monad{A,S}}`, that is, a function from outputs `A` to monads `Monad{A,S}` (which, remember, are also functions). The equation `ret(a)(s)  = (a,s)` could have alternatively been written `ret(a) = lambda s : (a,s)`; both say that `ret(a)` "returns" the output `a` and leave the state `s` unchanged, which corresponds to the expected definition.

The module for the operation `bind` involves monads with different output sorts:

```
(mod BIND {A :: TRIV, B :: TRIV,  S :: TRIV} is
protecting MONAD{A,S} .
protecting MONAD{B,S} .
protecting ARROW {A,Monad{B,S}} .
var m : Monad{A,S} .
var m' : Monad{B,S} .
var f :  Arrow{A, Monad{B,S}} .
vars a a' : A$Elt .
vars s s' : S$Elt .
 op bind : Monad{A,S} Arrow{A,Monad{B,S}} -> Monad{B,S} .
 ceq bind(m,f)(s) = (f(a) (s'))  if (a,s') := m(s) .
 op do_:=_in_ : A$Elt Monad{A,S} Monad{B,S} -> Monad{B,S} .
 eq (do a := m in m') = bind(m, (lambda a : m' )) .
 op _;;_ :  Monad{A,S} Monad{B,S} -> Monad{B,S} .
 eq (m ;; m')(s) = m' snd(m(s)) . endm)
```

The main operation `bind` takes a monad `m` of sort `Monad{A,S}`, a function `f` from `A` to `Monad{B,S}` and produces a monad `bind(m,f)` of sort `Monad{B,S}`. It first applies `m` to the current state `s`, which produces a pair `(a,s')`, then applies `f(a)` to `s'` ; cf. the equation `bind(m,f)(s) = (f(a)(s')) if (a,s') := m(s)`, whose condition is a *matching equation*, similar to a *let-in* construction in other languages.

A useful syntactical sugar involving `bind` is the construction `do a := m in m'` in which the result of computation `m` is "stored" in the "variable" `a`; the "value" of that variable can the be used in the subsequent computation `m'`. This construction gives the appearance of imperative code to purely functional code. Another imperative-like construction is the sequencing operation `m ;; m'`, which amounts to propagating the state (while discarding the output) computed by `m` to `m'`.

Thus, higher-order functions and state monads are available in Maude. We now show some simple examples of higher-order functional-imperative programs.

2.3. Example: Sorting Stacks

The running example in the rest of the paper is a program for sorting stacks.

Implementing stacks. The following Maude module constructs stacks using the constant `emptyStack` and an associative-unitary (AU) infix operation `_::_`. Operations `hd` and `tl` for the head and tail are defined as well, together with equations linking them to the constructor. An equality predicate `_===_` for stacks is given.

```
(fmod STACK {X :: TOTAL-ORDER} is
sort  Stack{X} .
subsort X$Elt < Stack{X} .
op emptyStack : -> Stack{X} .
op _::_ : Stack{X} Stack{X} -> Stack{X} [ctor assoc id: emptyStack] .
op hd : Stack{X} ~> X$Elt .
op tl : Stack{X} ~> Stack{X} .
op _===_ : Stack{X} Stack{X} ~> Bool .
vars p q r :  X$Elt .
var stk : Stack{X} .
eq tl(p :: stk) = stk .
eq hd(p :: stk) = p .
eq stk  ===  stk  = true .
endfm)
```

The module's parameter, `X`, is not a `TRIV` as before but a `TOTAL-ORDER`, a theory that declares a sort `Elt` and a total order `_<=_` on it. The *subsort declaration* `subsort X$Elt < Stack{X}` says that elements of `X$Elt` are stacks as well; also, the symbol `~>` indicates that the operators `hd` and `tl` are partial functions.

The state. We define the state to be a pair consisting of a natural number, which will be used to create fresh variables when such variables are needed, and a stack.

```
(fmod STATE {X :: TOTAL-ORDER}  is
protecting STACK{X} .
protecting NAT .
sorts State{X} .
*** the first argument is for generating fresh variables
op st : Nat Stack{X} -> State{X} .
endfm)
```

Basic monadic stack operations. As usual, we associate to stacks the operations `push` and `pop`. Being monadic operations, they shall have states as parameters and shall return pairs of output and state. The `push` operation pushes its argument on the stack and does not output anything - here, modelled by "outputting" `tt`, a constant of sort `Unit` defined in the imported module `UNIT`. The `pop` operation removes the top of the stack and outputs it using the stack operations `hd` and `tl`.

```
(fmod PUSH {X :: TOTAL-ORDER}  is
protecting UNIT . ...
op push : X$Elt -> Monad{Unit,State{X}} .
eq push(n) st(p, stk) = (tt , st(p, n :: stk)) .
endfm)
(fmod POP {X :: TOTAL-ORDER} is ...
op pop :  -> Monad{X,State{X}} .
eq pop (st(p, stk))  = (hd(stk) , st(p,tl(stk))) .
endfm)
```

The operation ***min****.* This monadic operation takes a stack and "moves" the smallest element of the stack (according to an order relation `leq` it takes as a parameter) to the top of the stack. It is defined as follows (Fig. 5): for empty stacks and one-element stacks the operation outputs `tt` and lets the stack in the state unchanged.

However, for stacks consisting of at least two elements, the operation first pops the stack and saves the output into a variable `x`, then recursively calls itself on the popped stack. Next, it pops the result and saves in a variable `y`. Finally, `x` and `y` are pushed back onto the stack so that the smallest of them goes to the top.

One can see, intuitively, that at the end one indeed obtains a stack where the smallest element is at the top and which has the same elements as the stack given as parameter. Later in the paper we shall prove this property formally.

Note the *matching conditions* in the third equation: `x` and `y` are created on-the-fly by a function `freshVar()` (not shown here) that takes the natural-number parameter we added to the state for this purpose. These matching equations are

```
(fmod SORT{X :: TOTAL-ORDER} is  ...
op min :  Arrow{X, Arrow{X, Bool}} -> Monad{Unit, State{X}} .
eq (min(leq) st(t,emptyStack)) = (tt,st(t,emptyStack)) .
eq (min(leq) st(t,n)) = (tt,st(t, n)) .
eq  min(leq)  st(t, n :: m :: stk) =
       (do x :=  pop in
          (min(leq) ;;
            do y :=  pop in
                 if (leq x) y  then
                  push(y) ;;
                  push(x)
                else
                  push(x) ;;
                  push(y)
                fi
          ))  st(t + 2, n :: m :: stk)
if x :=  freshVar(t) /\ y := freshVar(t + 1) ...
```

Figure 5: Module `SORT` (excerpt): operation `min`.

used to make the third equation *executable* by Maude; without them, `x` and `y` would be variables in the right-hand side of the equation that do not appear in its left-hand side. Such equations are rejected by Maude as being non-executable.

Note also the use of the builtin Maude `if-then-else-fi` construction. Here we are performing a shallow embedding of a language (with instructions `push`, `pop`, `do _:=_ in _`, `_;;_`, `min` etc) into Maude; that is, we are using the syntax and semantics of Maude to define the instructions of our embedded language, and we are allowed to use all Maude constructions, including `if-then-else-fi`, in programs of our language. This is in contrast to deep embeddings, which would amount to use Maude to define the syntax and semantics of a given language; then, programs in the language would only be allowed to use the defined syntax, excluding any other Maude construction (otherwise, they would not even parse).

The operation ***`sort`****.* This monadic operation (Fig. 6) is the one that performs the stack sorting according to an order `leq` it receivers as a parameter. The first two equations deal with the cases of empty and of one-element stacks. The interesting case is that of stacks with at least two elements, which is dealt with by the third equation. The previously defined operation `min` is first called to bring out the smallest element and put it on the top of the stack. This element is then popped and saved in a variable `x`. Next, the operation `sort` is recursively called on the

```
op sort : Arrow{X, Arrow{X, Bool}} -> Monad{Unit, State{X}} .
eq sort(leq) st(t,emptyStack) = (tt,st(t,emptyStack)) .
eq sort(leq) st(t,n) = (tt,st(t,n )) .
eq sort(leq) st(t, n :: m :: stk)  =
      (min(leq) ;;
       (do x := pop in (sort(leq) ;; push(x) ))
       )
       st(t + 1, n :: m :: stk)
if x := freshVar(t) .
endfm)
```

Figure 6: Module `SORT` (excerpt): operation `sort`.

popped stack, and finally the previously saved `x` is pushed back on the stack.

Here again one can see, intuitively, that at the end one gets a sorted stack that has the same elements as the stack one started with. Later in the paper we shall se how to prove this formally by executing a certain Maude command. We also prove in the paper that the command does indeed perform the expected verification.

Running the programs. In order to run the programs we instantiate them on a predefined Maude module `Nat<=` that provides the predefined sort `Nat` and the order `<=`. Now, `<=` is predefined as an operation: `_<=_ : Nat Nat -> Bool`. As such it cannot be passed as parameter to our higher-order monadic operations (which require it). We thus define a constant `ord` as a relation on `Nat` and define it to be extensionally equal to `<=`. The constant `ord` can then be passed as parameter.

```
(mod INSTANTIATE-SORT is
protecting SORT{Nat<=} .
op ord : -> Arrow{Nat<=, Arrow{Nat<=, Bool}} .
vars x y : Nat .
eq ord x y = x <= y .
op reverse : Arrow{Nat<=, Arrow{Nat<=, Bool}} ->
             Arrow{Nat<=, Arrow{Nat<=, Bool}} .
var leq :  Arrow{Nat<=, Arrow{Nat<=, Bool}} .
eq (reverse(leq) (x)) (y) = (leq y) x  .
endm)
```

Finally we define a function `reverse` that "reverses" a given order, and call `sort(ord)` and the `sort(reverse(ord))` on inputs to test their correctness:

```
Maude> (red sort(ord) st(0, 1 :: 4 :: 2 :: 8 :: 5 :: 7) .)
result Pair{Unit,State{Nat<=}} :
(tt, st(35, 1 :: 2 :: 4 :: 5 :: 7 :: 8))
Maude> (red sort(reverse(ord)) st(0, 1 :: 4 :: 2 :: 8 :: 5 :: 7) .)
result Pair{Unit,State{Nat<=}} :
(tt, st(35, 8 :: 7 :: 5 :: 4 :: 2 :: 1))
```

Maude returns the expected results instantaneously. Thus, we now have a shallow embedding of a language (with instructions `push`, `pop`, `do _:=_ in _`, `_;;_`, `min`, and `sort`) into Maude, which runs efficiently on given concrete inputs. In the next section we show how the language can be transformed in order to allow for symbolic execution of programs, i.e., running programs with symbolic inputs.

3. Symbolic Execution

In this section we show how one can symbolically execute programs such as those shown in the previous section. The approach is parametric in the definition of language whose operational semantics are defined using Reachability Logic formulas.

*3.1. Reachability Logic (*RL*)*

Several versions of Reachability Logic have been proposed in the last few years [3, 4, 5, 6]. Moreover, RL is built on top of *Matching Logic* (ML), which also exists in several versions [38, 39, 40]. (The situation is somewhat similar to the relationship between rewriting logic and the equational logics underneath it.) We adopt the recent *all-paths* interpretation of RL [6], and build it upon a minimal ML that is enough to express typical practically-relevant language semantics and properties of programs, and moreover is amenable to symbolic execution by rewriting.

The formulas of ML are called *patterns* and are defined below. Assume an algebraic signature Σ with a set S of sorts, including sorts *Bool* and *Cfg* (*configurations*). We write $T_{\Sigma,s}(Var)$ for the set of terms of sort s over a countable set *Var* of S-indexed variables, and $T_{\Sigma,s}$ for the set of ground terms of sort s.

Example 1. The sort *Cfg* of configurations in the examples from the previous section is `Pair{XuUnit,State}` where `XuUnit` is the smallest supersort of `X$Elt` and `Unit`. Sample *Cfg* terms are `(n,st(t,stk))` and `sort(leq)st(t,n :: m :: stk)` with sorted variables `leq:Arrow{X, Arrow{X, Bool}}`, `n:X$Elt`, `m:X$Elt`, `t:Nat`, and `stk:Stack{X}`.

We identify the *Bool*-sorted operations in Σ with a set Π of predicates.

Definition 1 (Pattern)**.** *A pattern is an expression of the form* $(\exists X)\pi\wedge\phi$*, where* $X \subset Var$*,* $\pi \in T_{\Sigma,Cfg}(X)$ *and* ϕ *is a First-Order Logic (*FOL*) formula over the first-order signature* (Σ, Π) *with free variables in* X*.*

We often denote patterns by φ and write $\varphi \triangleq (\exists X)\pi\wedge\phi$ to emphasise the quantified variables X, the *basic pattern* π, and ϕ, the *condition*. We let $FreeVars(\varphi)$ denote the set of variables freely occurring in a pattern φ, defined as usual (i.e., not under the incidence of a quantifier). We identify basic patterns π with patterns $(\exists\emptyset)\pi\wedge true$, and *elementary patterns* of the form $\pi\wedge\phi$ with patterns $(\exists\emptyset)\pi\wedge\phi$.

Example 2. The writing of patterns in Maude differs from their mathematical notation. Firstly, we enrich the structure of the sort `State{X}` with a third component of sort `Bool`, destined to encode pattern conditions. Secondly, in Maude there are no (explicit) existential quantifiers. We encode quantified variables by fresh variables, constructed with the function `freshVar()`. An example of quantified pattern with these conventions is the right-hand side of one of the equations for `sort`, where `f` plays the role of the condition and `x` of a quantified variable:

```
eq sort(leq) st(t, n :: m :: stk) =
       (min(leq) ;; (do x := pop in
             (sort(leq) ;;
              push(x) ))) st(t + 1, f, n :: m :: stk)
        if x := freshVar(t)
```

We now describe the semantics of patterns. We assume a model $\mathcal{M}$ of the algebraic signature Σ. In the case of our Maude examples, the model $\mathcal{M}$ is the initial model induced by the specification's equations and axioms. For sorts $s \in S$ we write $\mathcal{M}_s$ for the interpretation of the sort s. We call *valuations* the functions $\rho : Var \to \mathcal{M}$ that assign to variables values of a corresponding sort, and (concrete) *configurations* the elements in the carrier set $\mathcal{M}_{Cfg}$. An example of concrete configuration is the equivalence class w.r.t. all equations and axioms in our Maude specification for the ground term `(tt,st(0, false, emptyStack))` of sort *Cfg*.

Definition 2 (Pattern semantics)**.** *Given a pattern* $\varphi \triangleq (\exists X)\pi\wedge\phi$*,* $\gamma \in M_{Cfg}$ *a configuration, and* $\rho : Var \to \mathcal{M}$ *a valuation, the satisfaction relation* $(\gamma, \rho) \models \varphi$ *holds iff there exists a valuation* ρ' *with* $\rho'|_{Var\setminus X} = \rho|_{Var\setminus X}$ *such that* $\gamma = \rho'(\pi)$ *and* $\rho' \models \phi$ *(where the latter* $\models$ *denotes satisfaction in* FOL*, and* $\rho_{|Var\setminus X}$ *denotes the restriction of the valuation* ρ *to the set* $Var \setminus X$*).*

We now recall Reachability-Logic (RL) formulas, the transition systems that they induce, and their all-paths semantics [6] that we use in this paper.

Definition 3 (RL formula)**.** *An* RL *formula is a pair of patterns* $\varphi \Rightarrow \varphi'$*.*

Example 3. In the previous section we illustrated the shallow embedding of a language with instructions `do_:=_in_`, `_;;_`, `push`, `pop`, `min` and `sort` in Maude, whose semantics was defined using equations. We turn those equations into Maude *rewrite rules* and interpret the resulting rules as RL formulas. For example, the equation `sort(leq) st(t,f,emptyStack) = (tt,st(t,f,emptyStack))` becomes the rule `sort(leq) st(t,f,emptyStack) => (tt,st(t,f,emptyStack))`. Both equations and rules are applied by rewriting, but equations are destined to encode deterministic computations (such as concrete ones in our language), whereas rules encode possibly nondeterministic computations (such as symbolic ones).

Let $\mathcal{S}$ denote a fixed set of RL formulas, e.g., the semantics of a given language. We define the transition system defined by $\mathcal{S}$ and then the validity of RL formulas.

Definition 4 (Transition System defined by set $\mathcal{S}$ of RL formulas)**.** *The* transition system *defined by* $\mathcal{S}$ *is* $(\mathcal{M}_{Cfg}, \Rightarrow_{\mathcal{S}})$*, where* $\Rightarrow_{\mathcal{S}} = \{(\gamma, \gamma') \mid (\exists \varphi \Rightarrow \varphi' \in \mathcal{S})(\exists \rho)(\gamma, \rho) \models \varphi \wedge (\gamma', \rho) \models \varphi'\}$*. We write* $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ *for* $(\gamma, \gamma') \in \Rightarrow_{\mathcal{S}}$*. A state* γ *is* terminal *if there is no* γ' *such that* $\gamma \Rightarrow_{\mathcal{S}} \gamma'$*. A* path *is a sequence* $\gamma_0 \cdots \gamma_n$ *such that* $\gamma_i \Rightarrow_{\mathcal{S}} \gamma_{i+1}$ *for all* $0 \leq i \leq n-1$*. Such a path is* complete *if* γ_n *is terminal.*

Thus, the transition system induced by a set $\mathcal{S}$ of RL formulas contains all the possible computations of all possible programs in all possible contexts.

Definition 5 (Validity)**.** *An* RL *formula* $\varphi \Rightarrow \varphi'$ *is valid, written* $\mathcal{S} \models \varphi \Rightarrow \varphi'$*, if for all pairs* (γ_0, ρ) *such that* $(\gamma_0, \rho) \models \varphi$*, and all complete paths* $\gamma_0 \Rightarrow_{\mathcal{S}} \cdots \Rightarrow_{\mathcal{S}} \gamma_n$*, there exists* $0 \leq i \leq n$ *such that* $(\gamma_i, \rho) \models \varphi'$*.*

Note that the validity of RL formulas is only determined by finite, complete paths. Infinite paths, induced by nonterminating programs, are irrelevant. Thus, termination is assumed: as a program logic RL is a logic of partial correctness.

Assumption 1. *Hereafter,* RL *formulas have the form* $\pi_l \wedge \phi_l \Rightarrow (\exists Y)\pi_r \wedge \phi_r$ *such that* $FreeVars(\pi_r) \subseteq FreeVars(\pi_l) \cup Y$*,* $FreeVars(\phi_r) \subseteq FreeVars(\pi_l) \cup FreeVars(\pi_r)$*, and* $FreeVars(\phi_l) \subseteq FreeVars(\pi_l)$*.*

That is, the left-hand side is an elementary pattern, and the right hand side is a pattern, possibly with quantifiers; the remaining conditions prevent additional variables in right-hand sides and in conditions. Such formulas are typically expressive enough for expressing language semantics[2] and program properties.

[2]e.g., languages defined in the $\mathbb{K}$ framework: `http://k-framework.org`.

3.2. Language-Parametric Symbolic Execution

We now present symbolic execution, a program-analysis technique that consists in executing programs with symbolic inputs (e.g., a symbolic value x) instead of concrete inputs (e.g., 0) and in maintaining *path conditions* on the symbolic inputs.

We reformulate the language-independent symbolic execution approach already presented elsewhere [11], with notable simplifications (e.g., we do not use coinduction). The approach consists in transforming the semantics of a programming language so that, under reasonable restrictions, shown below, executing a program with the modified semantics amounts to executing the program symbolically. The approach also distinguishes between *builtin* (e.g., integers, booleans, ...) and *non-builtin* (e.g., various programming language constructs) sorts and operations. This distinction is important because it allows one to replace unification and narrowing (which are the natural operations in symbolic execution of rewrite theories, cf, e.g., [33]) with the much more efficient matching and rewriting, in a sound way.

Assumption 2. *There exists a* builtin subsignature $\Sigma^b \subsetneq \Sigma$. *The sorts and operations in* Σ^b *are* builtin*; all others are* non-builtin. *The sort Cfg is not builtin.*

Moreover, non-builtin operation symbols may only be subject to a (possibly empty) set of *linear, regular, and non-builtin* equations.

Definition 6. *An equation* $u = v$ *is: linear if both* u, v *are linear (a term is linear if any variable occurs in it at most once); regular if both* u, v *have the same set of variables; and non-builtin if both* u, v *only have non-builtin operations.*

Example 4. In our language we choose the builtin subsignature to be the module `STACK{X}`. That is, the builtin sorts are `Nat` and `Bool` (imported in the module), together with `X$Elt` and `State{X}` defined in the module. All the other sorts are non-builtin. One can verify that the equations remaining after the equation-to-rule transformation (cf. Example 3 above) are linear, regular and non-builtin.

In order to formulate the simulation result we now define the transition relation generated by the set of symbolic rules $\mathcal{S}^{\mathbf{s}}$. It is essentially rewriting modulo the congruence on $T_\Sigma(Var)$ induced by Assumption 2.

Let $Var^b \subset Var$ be variables of builtin sorts. We need the following technical assumption, which amounts to saying that variables in symbolic execution are builtin variables, and does not otherwise restrict the generality of our approach:

Assumption 3. *For every* $\pi_l \wedge \phi_l \Rightarrow (\exists Y)\pi_r \wedge \phi_r \in \mathcal{S}$, $\pi_l \in T_{\Sigma \setminus \Sigma^b}(Var)$, π_l *is linear, and* $Y \subseteq Var^b$.

The assumption can always be made to hold by replacing in π_l all non-variable terms in Σ^b and all duplicated variables by fresh variables, and by equating in the condition ϕ_l the new variables to the terms that they replaced.

Example 5. Some of the rules (formerly, equations) in our examples do no satisfy Assumption 3 and need to be transformed in order satisfy it. For example, the third rule for `sort` (cf. Figure 6) contains the builtin operation `_::_` on builtin sort `Stack{X}` in its left-hand side, which violates the constraint $\pi_l \in T_{\Sigma \setminus \Sigma^b}(Var)$. It can be made to comply to this constraint by (equivalently) writing it in the form

```
rl sort(leq) st(t, f, stk)  =>
     (min(leq) ;;
      (do x := pop in (sort(leq) ;; push(x) )))
st(t + 3, f, stk)
if n := freshVar(t) /\ m := freshVar(t + 1) /\
stk' := freshStkVar(t) /\ (n :: m :: stk' = stk) /\
x := freshVar(t + 2) .
```

After the transformation, the builtin operation `_::_` has been moved from the rule's left hand side to the condition, with additional fresh variables (to make the rule executable by Maude) and the additional condition `n :: m :: stk' = stk`. Such transformations are systematic and can be implemented in Maude itself.

In order to obtain symbolic execution additional transformations must be performed on rules. We define them formally and then illustrate them with examples.

Consider the signature Σ corresponding to a language definition. Let *Fol* be a new sort whose terms are all the FOL formulas, including quantifiers.

Let *Id* and *IdSet* be new sorts denoting identifiers and sets of identifiers, with a union operation $\_,\_$. Let $Cfg^{\mathfrak{s}}$ be a new sort, with constructor $(\exists \_)\_\wedge\_ : IdSet \times Cfg \times Fol \to Cfg^{\mathfrak{s}}$. Thus, patterns $(\exists X)\pi \wedge \phi$ correspond to terms $(\exists X)\pi \wedge \phi$ of sort $Cfg^{\mathfrak{s}}$ in the enriched signature and reciprocally. Consider also the following set of RL formulas, called the *symbolic version of* $\mathcal{S}$:

$$\mathcal{S}^{\mathfrak{s}} \triangleq \{(\exists \mathcal{X})\pi_l \wedge \psi \Rightarrow (\exists \mathcal{X}, Y)\pi_r \wedge (\psi \wedge \phi_l \wedge \phi_r) | \pi_l \wedge \phi_l \Rightarrow (\exists Y)\pi_r \wedge \phi_r \in \mathcal{S}\}$$

with ψ a new variable of sort *Fol*, and $\mathcal{X}$ a new variable of sort *IdSet*.

When rewriting with such rules over a pattern, the variable ψ matches the pattern's condition and is strengthened with the rule's conditions φ_l, φ_r; and the quantified variables of the RL formula are added to those of the pattern, possibly after renaming in order to avoid having the same variable quantified twice.

Example 6. The rule in Example 5 has the sole condition[3] `n :: m :: stk' = stk`. In the symbolic version of the rule, it is incorporated into the "path condition" `f`:

```
rl sort(leq) st(t, f, stk)  =>
     (min(leq) ;;
       (do x := pop in (sort(leq) ;; push(x) )))
st(t + 3, f and (n :: m :: stk' === stk) , stk)
if n := freshVar(t) /\ m := freshVar(t + 1) /\ stk' := freshStkVar(t)
/\ x := freshVar(t + 2) .
```

Most of the other rules in our Maude-embedded language are unchanged by this transformation since they are unconditional (i.e., both conditions φ_l, φ_r are *true*) and are not (explicitly) existentially unquantified. The one important exception is the Maude builtin `if-then-else-fi` construction that we have used in our language, which avoided the need to define a conditional instruction together with corresponding Boolean expressions, and allowed us to just use the existing constructions of the host Maude (in the spirit of shallow embedding). Had we defined a conditional instruction, the corresponding symbolic rules would be:

```
sif b then m1 else m2 fi st(p,f,stk) => m1 st(p,(f and b),stk)
sif b then m1 else m2 fi st(p,f,stk) => m2 st(p,(f and Not(b)),stk)
```

The global effect of the rules is that the current "path condition" `f` is either enriched with the instruction's condition `b` or with its negation `Not(b)`. We use `sif` ("symbolic if") instead of `if` to avoid confusion with Maude's builtin `if-then-else-fi` construction, and `Not` instead of the builtin `not` in order to avoid Maude's reduction of `not` to `xor` that makes resulting conditions unreadable.

The global interest of the above-described construction is that rewriting with the rules in $\mathcal{S}^{\mathfrak{s}}$ achieves a *mutual simulation* of rewriting with the rules in $\mathcal{S}$.

To comply with the definition of rewriting we need to extend the congruence $\cong$ to terms of sort $Cfg^{\mathfrak{s}}$ by $(\exists X)\pi_1 \wedge \phi \cong (\exists X)\pi_2 \wedge \phi$ iff $\pi_1 \cong \pi_2$.

Definition 7 (Relation $\Rightarrow_{\alpha^{\mathfrak{s}}}$)**.** *For* $\alpha^{\mathfrak{s}} \triangleq (\exists \mathcal{X})\pi_l \wedge \psi \Rightarrow (\exists \mathcal{X}, Y)\pi_r \wedge (\psi \wedge \phi_l \wedge \phi_r) \in \mathcal{S}^{\mathfrak{s}}$ *we write* $(\exists X)\pi \wedge \phi \Rightarrow_{\alpha^{\mathfrak{s}}} (\exists X, Y)\pi' \wedge \phi'$ if pattern $(\exists X)\pi \wedge \phi$ is rewritten by $\alpha^{\mathfrak{s}}$ to $(\exists X, Y)\pi' \wedge \phi'$*, i.e., there exists a substitution* σ' *on* $Var \cup \{\mathcal{X}, \psi\}$ *such that* $\sigma'((\exists \mathcal{X})\pi_l \wedge \psi) \cong (\exists X)\pi \wedge \phi$ *and* $\sigma'((\exists \mathcal{X}, Y)\pi_r \wedge (\psi \wedge \phi_l \wedge \phi_r) = (\exists X, Y)\pi' \wedge \phi'$.

Lemma 1 ($\Rightarrow_{\alpha^{\mathfrak{s}}}$ simulates $\Rightarrow_{\alpha}$)**.** *For all configurations* $\gamma, \gamma' \in \mathcal{M}_{Cfg}$*, all patterns* φ *with* $FreeVars(\varphi) \subseteq Var^b$*, and all valuations* ρ*, if* $(\gamma, \rho) \models \varphi$ *and* $\gamma \Rightarrow_{\alpha} \gamma'$ *then there exists* φ' *with* $FreeVars(\varphi') \subseteq Var^b$ *such that* $\varphi \Rightarrow_{\alpha^{\mathfrak{s}}} \varphi'$ *and* $(\gamma', \rho) \models \varphi'$.

[3] matching equations do not count as conditions as they just assist the matching process.

As a consequence, any concrete execution ($\Rightarrow_{\mathcal{S}}$) such that the initial configuration satisfies a given initial pattern φ is simulated by a symbolic execution ($\Rightarrow_{\mathcal{S}^\mathsf{s}}$) starting in φ. A simulation of the *reverse* relation $\Leftarrow_{\mathcal{S}^\mathsf{s}}$ by the relation $\Leftarrow_{\mathcal{S}}$ holds:

Lemma 2 ($\Leftarrow_\alpha$ simulates $\Leftarrow_{\alpha^\mathsf{s}}$)**.** *For all all patterns φ, φ' such that $\varphi \Rightarrow_{\alpha^\mathsf{s}} \varphi'$, for all configurations $\gamma' \in \mathcal{M}_{Cfg}$ and valuations ρ such that $(\gamma', \rho) \models \varphi'$, there exists a configuration $\gamma \in \mathcal{M}_{Cfg}$ such that $(\gamma, \rho) \models \varphi$ and $\gamma \Rightarrow_\alpha \gamma'$.*

Thus, a symbolic execution whose final pattern is satisfied by a given configuration corresponds to a concrete execution ending in that configuration. Note that the simulation of $\Rightarrow_{\mathcal{S}^\mathsf{s}}$ by the relation $\Rightarrow_{\mathcal{S}}$ does not hold in general: indeed, some symbolic executions do not correspond to any concrete executions at all. Such symbolic executions are called *unfeasible* and occur, in our framework, when rewriting generates patterns $(\exists X)\pi \wedge \phi$ with unsatisfiable conditions ϕ.

The notion of *derivative* is about symbolic successors of patterns by rules:

Definition 8 (Derivatives)**.** *We set $\Delta_\alpha(\varphi) = \varphi'$, where φ' is uniquely defined (up to variable names) by the transition $\varphi \Rightarrow_{\alpha^\mathsf{s}} \varphi'$, and $\Delta_{\mathcal{S}}(\varphi) = \{\Delta_\alpha(\varphi) | \alpha \in \mathcal{S}\}$.*

Example 7. We illustrate the symbolic execution of stack-sorting for stacks of, say, three elements. This is achieved with the Maude command shown below:

```
Maude> (search sort(ord) st(0, true, p :: q :: r ) =>!
        (tt,st(t:Nat, f:Bool,stk:Stack{X}))
        such that f:Bool =/= false .)
Solution 1
f:Bool --> p:X$Elt <= q:X$Elt and q:X$Elt <= r:X$Elt and (...);
t:Nat --> 18 ;
stk:Stack‘{X‘} --> p:X$Elt :: q:X$Elt :: r:X$Elt
...
Solution 6
f:Bool --> Not(p:X$Elt <= r:X$Elt)and Not(p:X$Elt <= q:X$Elt)and
Not(q:X$Elt <=  r:X$Elt) and (...);
t:Nat --> 18 ;
stk:Stack‘{X‘} --> r:X$Elt :: q:X$Elt :: p:X$Elt
```

The `search` command computes all irreducible (`=>!`) terms reachable by applying `sort(ord)` to a state consisting of fresh-variable counter initialised to 0, initial path condition `true`, and three-element stack `p :: q :: r`. Moreover, in the obtained irreducible terms only those with path condition `f` that does not evaluate to `false` are kept. The path conditions are not shown completely; the conjuncts `(...)` denote equalities involving the fresh variables (standing for existentially

quantified variables) introduced by the rewriting. Existentially quantifying those variables over whole path conditions reduces them to the path conditions fragments shown. There are 6 (= 3!) solutions as expected, of which only the first and last are shown. The orders (`ord` a.k.a. `<=`) between `p`, `q` and `r` in the path conditions are consistent with the fact that the stacks in the reported solutions are sorted.

4. Program Verification

Example 7 can be seen as a proof by symbolic execution that our procedure `sort(ord)` sorts stacks of three elements w.r.t. the relation `ord`. For stacks of an arbitrary number of elements, symbolic execution, although useful, is not enough, because it amounts to building an infinite tree. In this section we show how such infinite trees can be made finite, thereby achieving full formal verification.

Example 8. We first present an example to help the intuition. We are going to verify the fact that the procedure `min(leq)` from Section 2 transforms a stack `q :: stk` containing at least one element, into a stack `q' :: stk'` having the same elements (with same number of occurrences) as `q :: stk`, and whose first element `q'` is the least element (w.r.t. the order `leq`) in the stack `q' :: stk'`.

```
crl min(leq) st(t, true, true, (q :: stk)) =>
 (tt, st(t', true,
  (leq q' least(leq,q' :: stk')) and sameElements(q :: stk,q' :: stk'),
  q' :: stk')))
 if t' := freshNatVar(t)/\q' := freshVar(t)/\stk' := freshStackVar(t) .
```

As already seen above the fresh variables in the condition are actually existentially quantified variables, here, of types `Nat`, `X$Elt` and `Stack{X}` respectively. The functions `least` and `sameElements` return, respectively, the least element w.r.t. an order `leq` in a stack, and the truth value of a predicate stating that two stacks have the same elements with same number of occurrences.

The above formula thus specifies a property of the `min` function. It turns out that it can also be used to "prove" itself by symbolically executing itself (properly transformed to satisfy Assumption 3) together with the other rules in the definition of the `min` function. For this we have added to the constructor `st` of the sort `State{X}` a fourth component (placed in second position, just after the fresh-variable counter `t`). It is a Boolean that "forces" the application of other rules of the `min` function, before the last one can be applied. In this way vicious circular reasoning is avoided. The proof itself consists in a `search` command that attempts to find counterexamples to the property: irreducible

symbolic configurations that either contain a stack that does not have the expected form `q' :: stk'`, or whose condition does not imply the expected condition `(leq q' least(leq,q' :: stk')) and sameElements(q :: stk,q' :: stk')`.

For stacks of at least two elements the `search` command and the output are:

```
Maude> (search (min(ord) st(0,false,true, p1 :: q1 :: stk1)) =>! pp
          such that not ((expectedPattern(pp)) or
          not  (getPathCondition(pp)
                 Implies
                expectedCondition(ord,pp,p1 :: q1 :: stk1))).)
No solution.
```

A similar command, with same output, is used for stacks of one element. Since solutions correspond to counterexamples, and none are found, the RL formula is (claimed) valid. Of course, this is an informal argument; it is formalised below. The functions `expectedPattern`, `getPathCondition` and `expectedCondition` are equationally defined; their names are self-explanatory. In order to obtain the above expected result we have used several equations on the builtin types as simplification which are inductive consequences of the definitions of `least` and `sameElements`. This can be proved, for example, using Maude's inductive theorem prover ITP[4].

The above `search` command thus symbolically executes the left-hand side of the formula under proof, using the rules defining the `min` function and the formula under proof itself, starting with the former in order to avoid vicious circular reasoning. In the process the `search` command builds a *tree*[5] whose nodes are patterns; and checks certain conditions on its leaves, i.e., patterns without successors. Note that unfeasible program paths (constructed by symbolic execution) correspond to unsatisfiable leaves in the tree; since these imply any other pattern, verification remains sound. We now generalise and formally establish the soundness of the approach by defining a tree-construction procedure that corresponds to the Maude `search` command.

Before we describe the procedure we introduce its components. The procedure assumes a set of RL formulas $\mathcal{S}$ (the semantical rules of a programming language) and a set of RL formulas $\mathcal{G}$ under proof, called *circularities* in RL-based verification.

A partial order $<$ on $\mathcal{S} \cup \mathcal{G}$. During symbolic execution, circularities can be symbolically applied "in competition with" rules in the semantics. When this is the

[4]Currently available at `http://maude.cs.uiuc.edu/tools/itp/`.

[5]In general, `search` builds a graph, by looping back if a newly reached node equals one that was already encountered. However in our case new nodes are always different from already encountered ones due to the fresh-variable counter that always grows.

0: $G = (N \triangleq \{\pi \wedge \phi\} \cup \Delta_{\mathcal{S}}(\pi \wedge \phi), E \triangleq \{\pi \wedge \phi \xrightarrow{\alpha} \Delta_\alpha(\pi \wedge \phi) | \alpha \in \mathcal{S}\})$, $Fail \leftarrow false$, $New \leftarrow \Delta_{\mathcal{S}}(\pi \wedge \phi)$

1: **while** not $Fail$ and $New \neq \emptyset$

2: **choose** $\varphi_n \triangleq (\exists X_n)\pi_n \wedge \phi_n \in New$; $New \leftarrow New \setminus \{\varphi_n\}$

3: **if** $match_{\cong}(\pi_n, \pi') = \emptyset$ **then**

4: **if** $\bigvee_{\alpha \in min(<)} implication(\varphi_n, lhs(\alpha)) = true$ **then**

5: **forall** $\alpha \in min(<)$

6: $New \leftarrow New \cup \{\Delta_\alpha(\varphi_n)\}$; $E \leftarrow E \cup \{\varphi_n \xrightarrow{\alpha} \Delta_\alpha(\varphi_n)\}$

7: $N \leftarrow N \cup New$

8: **else** $Fail \leftarrow true$ **endif**

9: **elseif** not $implication(\varphi_n, (\exists Y)\pi' \wedge \phi')$ **then** $Fail \leftarrow true$ **endif**.

Figure 7: Tree construction. $match_{\cong}()$ is matching modulo the non-builtin axioms (cf. Section 3.2), and $implication()$ is the object of Definition 9.

case priority is given to circularities, because if both the circularities and the rules in the semantics can be applied then the latter will typically generate infinite symbolic executions, compromising the procedure's termination. To make this precise we use a partial order relation $<$ on $\mathcal{S} \cup \mathcal{G}$. We use the following notations. Let $lhs(\alpha)$ denote the left-hand side of a formula α. Let $\mathcal{G} < \mathcal{S}$ denote the fact that for all $g \in \mathcal{G}$ and $\alpha \in \mathcal{S}$, $g < \alpha$, and $min(<)$ denote the $<$-minimal elements of $\mathcal{S} \cup \mathcal{G}$.

Assumption 4. *We assume a partial order relation $<$ on $\mathcal{S} \cup \mathcal{G}$ satisfying: $\mathcal{G} < \mathcal{S}$, and for all $\alpha' \in \mathcal{S}$ and pairs (γ, ρ), if $(\gamma, \rho) \models lhs(\alpha')$ then there exists $\alpha \in min(<)$ such that $(\gamma, \rho) \models lhs(\alpha)$.*

This gives circularities priority over rules in the semantics that can be applied in competition with them. In our example the priorities did not require a partial order because in the only situation when the circularity could be in competition with another rule, the latter is not applied by using a simple technical trick.

Implication between patterns. Our tree-construction procedure uses a test of implication between patterns, which satisfies the following definition.

Definition 9 (Implication)**.** *An implication test is a function that, given patterns φ, φ', returns true if for all pairs (γ, ρ), if $(\gamma, \rho) \models \varphi$ then $(\gamma, \rho) \models \varphi'$.*

The tree construction. We are now ready to present the procedure in Fig. 7. The procedure takes as input an RL formula $\pi \wedge \phi \Rightarrow (\exists Y)\pi' \wedge \phi'$ from a set $\mathcal{G}$ of RL formulas, and a set $\mathcal{S}$ of RL formulas with an order $<$ on $\mathcal{S} \cup \mathcal{G}$ as discussed earlier in this section. It builds a tree (N, E) with N the set of nodes (initially consisting of the initial pattern $\{\pi \wedge \phi\}$ and of its $\mathcal{S}$-derivatives) and E the set of edges (initially connecting the initial node and its $\mathcal{S}$-derivatives). It uses two variables to control a while loop: a Boolean variable *Fail* (initially *false*) and a set of nodes *New* (initially containing the $\mathcal{S}$-derivatives of the initial node).

At each iteration of the while loop, a node $\varphi_n \triangleq (\exists X_n)\pi_n \wedge \phi_n$ is taken out from *New* (line 2) and it is checked whether there is a matcher modulo $\cong$ (cf. Section 3.2) of π' onto π_n (line 3). If this is the case, then π_n is an instance of the basic pattern π', and the procedure goes to line 9 to check whether φ_n "as a whole" is included in $(\exists Y)\pi' \wedge \phi'$. If this is not the case, then this indicates a terminal configuration that does not satisfy the right-hand side of the formula under proof; *Fail* is reported, which terminates the execution of the procedure. However, if the test at line 3 indicated that π_n is not an instance of the pattern π', then another implication test is performed (line 4): whether there exists a minimal element in $\mathcal{S}$ (i.e., from the language's semantics, or among the circularities) whose left-hand side includes φ_n. If this is not the case then procedure terminates again with *Fail* = *true*.

If, however, the implication test at line 4 succeeds then all symbolic successors φ'_n of φ_n by minimal elements α in $<$ are computed, and an edge from the current node φ_n to every new node, labelled by the artifact that generated it, is created.

The tree-construction procedure does not terminate in general, since the verification of RL formulas is undecidable. However, if it terminates with *Fail* = *false*:

Theorem 1 (Soundness)**.** *Consider a set of* RL *formulas* $\mathcal{S} \cup \mathcal{G}$*. If for all* $g \in \mathcal{G}$ *the procedure in Figure 7 terminates with Fail = false then* $\mathcal{S} \models \mathcal{G}$*.*

Theorem 1 uses the following (and last) assumptions on RL formulas, where for a pattern φ the notation $[\![\varphi]\!]$ denotes the set $\{\gamma \mid (\exists \rho)(\gamma, \rho) \models \varphi)\}$:

Assumption 5. *All rules* $\varphi_l \Rightarrow \varphi_r \in \mathcal{S}$ *have the following properties:*

1. *for all* (γ, ρ) *such that* $(\gamma, \rho) \models \varphi_l$ *there exists* γ' *such that* $(\gamma', \rho) \models \varphi_r$*.*
2. $[\![\varphi_l]\!] \cap [\![\varphi_r]\!] = \emptyset$*.*

The first of the above assumptions says that if the left-hand side matches a configuration then there is nothing in the right-hand side preventing the application. This property is called *weak well-definedness* in [6] and is shown there to be a necessary condition for obtaining a sound proof system for RL. The second

condition says that the left and right-hand sides of rules cannot share instances; this is quite natural since rules are meant to describe progress in a computation.

Remark 1. *Assumption 5.1 implies* $[\![\varphi_l]\!]$ *has no terminal configurations.*

To conclude this section we show that the function `min` for stacks of at least two elements (of the form `p1 :: q1 :: stk1`) does satisfy its RL specification (for stacks of one element the result is trivial). This was written earlier in the section:

```
crl min(leq) st(t, true, true, (q :: stk)) =>
(tt, st(t', true,
(leq q' least(leq,q' :: stk')) and sameElements(q :: stk,q' :: stk'),
q' :: stk')))
if t' := freshNatVar(t)/\q' := freshVar(t)/\stk' := freshStackVar(t)
```

The `search` command for proving this rule (interpreted as an RL formula) was:

```
(search (min(ord) st(0,false,true, p1 :: q1 :: stk1)) =>! pp
such that not ((expectedPattern(pp)) or
not (getPathCondition(pp)
Implies expectedCondition(ord,pp,p1 :: q1 :: stk1))).)
```

The command first applies rules in the definition of `min` for stacks of two elements. This is ensured by the Boolean flag used to prevent the application of circularities right from the beginning, and corresponds to the initialisation step of our tree-construction procedure (Fig. 7). Thus, the initial set of nodes (as computed by rewriting in Full Maude and slightly edited for readability) is the singleton term:

```
(do v3 := pop in
  sif p1 <= v3 then push(v3);; push(p1) else
  push(p1);; push(v3) fi)
  snd(min(ord)
   st(4,true,(v0 :: v1 :: vstk0)===(p1 :: q1 :: stk1),
   q1 :: stk1))
```

That is, `min` still needs to be recursively applied to the smaller stack `q1 :: stk1`, and the result `v3` of popping the initial stack `p1 :: q1 :: stk1` will be pushed on the result of the recursive call. The `snd` pair-destructor is a side effect of monads, and the equality of stacks in the condition is induced by the fact that rules are applied symbolically, i.e., after the rule transformation described in Section 3.2.

The circularity, i.e., the above RL formula under proof, can now be applied, and two terminal patterns are generated, which correspond to the variables `x` and `y` in `min` being pushed on the stack in one order or the other. Fresh variables have been generated in the process, and path conditions have been accumulated.

```
(tt,st(vnat4,true,p1 <= v5 and
    v5 <= least(ord,v5 :: vstk5)and
    (v4 :: vstk4 === q1 :: stk1 and (v0 :: v1 :: vstk0)===
    p1 :: q1 :: stk1 and sameElements(q1 :: stk1,v5 :: vstk5),
    p1 :: v5 :: vstk5))

(tt,st(vnat4,true,Not(p1 <= v5)and
    v5 <= least(ord,v5 :: vstk5) and
    (v4 :: vstk4)=== q1 :: stk1 and (v0 :: v1 :: vstk0)===
     p1 :: q1 :: stk1 and  sameElements(q1 :: stk1,v5 :: vstk5),
    v5 :: p1 :: vstk5))
```

One can see that these patterns satisfy `expectedPattern()` (i.e., both resulting stacks `p1 :: v5 :: vstk5` and `v5 :: p1 :: vstk5` are nonempty). This corresponds to jumping at line 9 in the procedure in Fig. 7. The patterns also both satisfy `getPathCondition() Implies expectedCondition ()` (i.e. the path condition implies the stacks are ordered according to `ord`). Thus, *implication* holds between each of the above patterns and the final expected one, hence, the procedure terminates with *Fail* = *false*. This is mirrored by that fact that the `search` command, which implements the tree construction, terminates with `No solution.`

Using the soundness result (Th. 1) we conclude that the function `min` does indeed satisfy its RL specification. A similar reasoning allows to prove the following formula, which says that the function `sort` returns a sorted stack with same elements as its input (the predicate `isSorted` is equationally specified in Maude):

```
crl sort(leq) st(t, true, f, stk) => (tt,st(t',true,f and
 isSorted(leq,stk') and sameElements(stk,stk')),stk'))
if t' := freshNatVar(t) /\ stk' :=  freshStackVar(t) .
```

5. Program Equivalence

Using symbolic execution we were able to verify programs with respect to RL formulas. We now show that symbolic execution and program verification enable us to prove equivalence between programs as well. Specifically, we are interested in *weak equivalence*, which says that two programs are equivalent if whenever presented with the same inputs, if both programs terminate then they compute the same outputs. This kind of equivalence is adapted for deterministic programs - each terminating computation on a given input produces a unique output.

We formalise this notion in a RL setting, and illustrate it by proving the equivalence of `sort(ord)` and `sort(reverse(ord));; rev` where `rev` is a program that reverses a stack and `_;;_` is the monadic sequencing operation.

We denote by $\Rightarrow^*_{\mathcal{S}}$ the reflexive-transitive closure of the relation $\Rightarrow_{\mathcal{S}}$. We write $\gamma_1 \Rightarrow^!_{\mathcal{S}} \gamma_2$ whenever the configuration γ_2 has no successor in the relation $\Rightarrow_{\mathcal{S}}$. We write $\varphi_1 \Rightarrow^!_{\mathcal{S}^\mathsf{s}} \varphi_2$ whenever $[\![\varphi_2]\!]$ contains such a terminal configuration.

Definition 10 (Confluence)**.** *$\mathcal{S}$ is confluent if for all distinct configurations γ, γ_1, γ_2 s.t. $\gamma \Rightarrow^*_{\mathcal{S}} \gamma_1$ and $\gamma \Rightarrow^*_{\mathcal{S}} \gamma_2$ there is γ' such that $\gamma_1 \Rightarrow^*_{\mathcal{S}} \gamma'$ and $\gamma_2 \Rightarrow^*_{\mathcal{S}} \gamma'$.*

Remark 2. *If $\mathcal{S}$ is confluent then for all configurations γ, the relation $\gamma \Rightarrow^!_{\mathcal{S}} \gamma'$ uniquely defines the configuration γ'.*

The above remark ensures that the following definition is sound whenever $\mathcal{S}$ is confluent (a hypothesis that we shall assume in the rest of this section):

Definition 11. *$last(\gamma) = \gamma'$ whenever $\gamma \Rightarrow^!_{\mathcal{S}} \gamma'$.*

Definition 12 (Program Equivalence)**.** *Consider an equivalence $\sim$ on $\mathcal{M}_{Cfg}$. For configurations γ_1, γ_2, we write $\gamma_1 \equiv \gamma_2$ iff $\gamma_1 \sim \gamma_2$ and (both $last(\gamma_1), last(\gamma_2)$ exist implies $last(\gamma_1) \sim last(\gamma_2)$). The relation $\sim$ is a (weak) program equivalence if for all pairs γ_1, γ_2, $\gamma_1 \sim \gamma_2$ implies $\gamma_1 \equiv \gamma_2$.*

The equivalence is called "weak" because it assumes that both $last(\gamma_1)$ and $last(\gamma_2)$ exist. Other versions of equivalence, which relax this constraint, have been defined in the literature, but we are not here concerned by them.

The relations $\sim$ and $\equiv$ can be lifted to patterns as well:

Definition 13 ($\sim$ and $\equiv$ on patterns)**.** *For patterns φ_1, φ_2 we write $\varphi_1 \sim \varphi_2$ (resp. $\varphi_1 \equiv \varphi_2$) if for all configurations γ_1, γ_2 and valuation ρ such that $(\gamma_1, \rho) \models \varphi_1$ and $(\gamma_2, \rho) \models \varphi_2$ the relation $\gamma_1 \sim \gamma_2$ (resp. $\gamma_1 \equiv \gamma_2$) holds.*

The following lemma gives sufficient conditions for equivalence on patterns (and thus, on the program configurations that they denote):

Lemma 3. *Consider two patterns φ_1, φ_2 such that $\varphi_1 \sim \varphi_2$. Then, $\varphi_1 \equiv \varphi_2$ if for all patterns φ'_1, φ'_2 such that $\varphi_1 \Rightarrow^!_{\mathcal{S}^\mathsf{s}} \varphi'_1$ and $\varphi_2 \Rightarrow^!_{\mathcal{S}^\mathsf{s}} \varphi'_2$, $\varphi'_1 \sim \varphi'_2$ holds.*

Lemma 3 suggests that it is enough to compute all symbolic successors of two patterns that contain some terminal configuration in order to check their equivalence. Unfortunately there are in general infinitely many such symbolic successors even if all the program configurations denoted by the patterns terminate (i.e., each execution length is finite but the set of all executions lengths is unbounded).

However, this situation can be improved if we prove some RL properties on the programs we want to prove equivalent since we can then use the properties as rewrite rules to obtain over-approximations of reachable symbolic successors. The following lemma formalises this intuition.

Lemma 4. *If* $\mathcal{S} \models \mathcal{G}$ *and* $\varphi \Rightarrow^{!}_{\mathcal{S}^s} \varphi'$ *then there is* φ'' *such that implication*(φ', φ'') *and* $\varphi \Rightarrow^{!}_{\mathcal{S}'^s} \varphi''$ *with* $\mathcal{S}' \triangleq \min(<)$.

That is, it is enough to consider the rules in $\min(<)$, which replace some of the rules in $\mathcal{S}$ with (proved) circularities $\mathcal{G}$. The proof of Lemma 3 is easily redone if over-approximations φ''_i of φ'_i $(i = 1, 2)$ are used instead of φ'_i. Combined with Lemma 4 this gives us a practical approach to prove pattern equivalence $\varphi_1 \equiv \varphi_2$:

- prove $\mathcal{S} \models \mathcal{G}_i$ and determine the relations $<_i$, for $i = 1, 2$;
- compute all symbolic successors φ''_i of φ_i such that $\varphi_i \Rightarrow^{!}_{\mathcal{S}'^s} \varphi''_i$, for $i = 1, 2$;
- for each pairs of such successors φ''_1 of φ_1 and φ''_2 of φ_2, show $\varphi''_1 \sim \varphi''_2$.

Going back to our example on proving the equivalence of computing `sort(ord)` and `sort(reverse(ord));; rev` an stacks: the equivalence $\sim$ is the equality of stacks given to the two programs, and the minimal elements of the relation $<$ consist of the single formula proved at the end of Section 4:

```
crl sort(leq) st(t, true, f, stk) => (tt,st(t',true,f and
 isSorted(leq,stk') and sameElements(stk,stk')),stk'))
if t' := freshNatVar(t) /\ stk' :=  freshStackVar(t) .
```

To give priority to it we remove all the other rules defining `sort`; this is sound because had we implemented priorities, e.g., at the metalevel, the removed rules would not have been executed. In the resulting module we add a new conditional rule `(p1,p2) => (p'1,p'2) if p1 => p'1 /\ p2 => p'2` stating that a pattern pair `(p1,p2)` is rewritten in one step into another pattern pair `(p'1,p'2)` whenever the composing patterns are rewritten component-wise, possibly in several steps; and we use `search`:

```
(search ((sort(ord)st(0,true, true, stk)),
         (sort(reverse(ord)) ;; rev)st(1 ,true, true, stk)) =>*
         ((tt,st(t1,f'1,f1, stk1),(tt,st(t2,f'2,f2,stk2))) such that
          not ((f1 and f2) Implies (stk1 === stk2)) .)
No solution.
```

That is, starting from an initial pattern pair where the components contain the same stack `stk`, we are looking for the only possible form of reachable pair of patterns that may contain terminal configurations *and* whose conjunction of path conditions `f1 and f2` does *not* imply the equality of the resulting stacks `stk1 === stk2`. Since the `search` command returns no solution, we obtain that in all patterns (with stacks `stk1`, `stk2`) possibly containing a terminal configuration and

reachable from the initial patterns (both containing the same stack `stk`) it must be the case thatthe equality `stk1 === stk2` holds, which, by the results established in this section, proves the weak equivalence of our programs.

Remark 3. *We started with 0 in one of the programs and 1 in the other one is in order to avoid interference in the variable names that are created during execution. Per the discussion just above the last* `search` *command there is only one rule defining* `sort` *to be called - the circularity, reproduced above - which does not have recursive calls and only creates two variables - a* `Nat` *and a* `Stack` *- using the natural number provided in the input. By starting with distinct natural numbers the resulting four variable names are distinct, thus, name interference is avoided.*

Like in the case of program verification we have used here equationally defined functions on the builtin sort `Stack{X}`, together with equations for simplification, that are inductive consequences of the function's definition. Actually proving them using, e.g., Maude's ITP theorem prover remains a matter of future work.

6. Conclusion and Future Work

In this paper we have shown that Maude is an all-in-one framework where one can write, execute and verify higher-order functional-imperative programs. We used (Full) Maude's module system and reflective features to incorporate higher-order functions and state monads in Maude, thereby obtaining a shallow embedding of the desired higher-order imperative functional language. We then adapted existing techniques for symbolic execution and Reachability-Logic verification (currently used essentially for deep-embedded language definitions) to sample programs in the resulting shallow-embedded language. We have also shown that program equivalence can be proved using symbolic execution and program verification.

There are several developments that we leave as future work. First, domain-specific properties on builtins (inductive consequences of definitions) that we use as simplification rules could be proved using Maude's Inductive Theorem Prover ITP, thereby eliminating the need to assert them as axioms. Technical issues regarding interactions between Full Maude and ITP need to be resolved beforehand.

Second, our current approach for RL formula verification (using Maude's search command at base level) uses some ad-hoc technical tricks to implement the formally proved sound procedure we present in the paper. The procedure itself requires a meta-level implementation. Being already in Full Maude, a meta-level implementation resides at the meta-meta level with respect to Maude's rewriting engine, which may raise questions of efficiency that need to be dealt with.

Third and last, higher-order functions and state monads can also be used in other Maude developments. We are investigating an extension of Maude's current

object-oriented modules, in which classes and objects would have actual methods (i.e., constants of higher-order types) and methods would have imperative code.

7. References

[1] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All about Maude – a high-performance logical framework: how to specify, program and verify systems in Rewriting Logic, Springer-Verlag, 2007.

[2] The haskell language, `http://www.haskell.org`.

[3] G. Roşu, A. Ştefănescu, Towards a unified theory of operational and axiomatic semantics, in: Proceedings of the 39th International Colloquium on Automata, Languages and Programming (ICALP'12), Vol. 7392 of Lecture Notes in Computer Science, Springer, 2012, pp. 351–363.

[4] G. Roşu, A. Ştefănescu, Checking reachability using matching logic, in: Proceedings of the 27th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12), ACM, 2012, pp. 555–574.

[5] G. Roşu, A. Ştefănescu, Ş. Ciobâcă, B. M. Moore, One-path reachability logic, in: Proceedings of the 28th Symposium on Logic in Computer Science (LICS'13), IEEE, 2013, pp. 358–367.

[6] A. Ştefănescu, Ş. Ciobâcă, R. Mereuţă, B. M. Moore, T. F. Şerbănuţă, G. Roşu, All-path reachability logic, in: Proceedings of the Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications (RTA-TLCA'14), Vol. 8560 of LNCS, Springer, 2014, pp. 425–440.

[7] The Coq proof assistant, `http://coq.inria.fr`.

[8] W. Swierstra, A Hoare logic for the state monad, in: S. Berghofer, T. Nipkow, C. Urban, M. Wenzel (Eds.), Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings, Vol. 5674 of Lecture Notes in Computer Science, Springer, 2009, pp. 440–451.

[9] A. Ştefănescu, D. Park, S. Yuwen, Y. Li, G. Roşu, Semantics-based program verifiers for all languages, in: Proceedings of the 31th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16), ACM, 2016.

[10] The $\mathbb{K}$ semantic framework, http://www.kframework.org.

[11] D. Lucanu, V. Rusu, A. Arusoaie, A Generic Framework for Symbolic Execution: a Coinductive Approach, Journal of Symbolic Computation-doi:10.1016/j.jsc.2016.07.012.
URL https://hal.inria.fr/hal-01238696

[12] V. Rusu, D. Lucanu, T. Serbanuta, A. Arusoaie, A. Stefanescu, G. Rosu, Language definitions as rewrite theories, J. Log. Algebr. Meth. Program. 85 (1) (2016) 98–120.
URL http://dx.doi.org/10.1016/j.jlamp.2015.09.001

[13] Ştefan Ciobâcă, D. Lucanu, V. Rusu, G. Rosu, A language-independent proof system for full program equivalence, Formal Asp. Comput. 28 (3) (2016) 469–497.
URL http://dx.doi.org/10.1007/s00165-016-0361-7

[14] A. Arusoaie, D. Lucanu, V. Rusu, Symbolic execution based on language transformation, Computer Languages, Systems & Structures 44 (2015) 48–71.
URL http://dx.doi.org/10.1016/j.cl.2015.08.004

[15] D. Lucanu, V. Rusu, Program equivalence by circular reasoning, Formal Asp. Comput. 27 (4) (2015) 701–726.
URL http://dx.doi.org/10.1007/s00165-014-0319-6

[16] J. C. King, Symbolic execution and program testing, Communications of the ACM 19 (7) (1976) 385–394.

[17] C. S. Păsăreanu, N. Rungta, Symbolic PathFinder: symbolic execution of Java bytecode, in: International Conference on Automated Software Engineering, ASE'10, ACM, 2010, pp. 179–180.

[18] P. Godefroid, N. Klarlund, K. Sen, DART: directed automated random testing, in: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, ACM, 2005, pp. 213–223.

[19] K. Sen, D. Marinov, G. Agha, CUTE: a concolic unit testing engine for C, in: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, ESEC/FSE-13, ACM, New York, NY, USA, 2005, pp. 263–272.
URL http://doi.acm.org/10.1145/1081706.1081750

[20] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, D. R. Engler, EXE: automatically generating inputs of death, in: ACM Conference on Computer and Communications Security, 2006, pp. 322–335.

[21] J. de Halleux, N. Tillmann, Parameterized unit testing with Pex, in: Tests and Proofs, Second International Conference, Vol. 4966 of Lecture Notes in Computer Science, Springer, 2008, pp. 171–181.

[22] C. Cadar, D. Dunbar, D. Engler, Klee: unassisted and automatic generation of high-coverage tests for complex systems programs, in: Proc. 8th USENIX conference on Operating systems design and implementation, OSDI'08, 2008, pp. 209–224.
URL http://dl.acm.org/citation.cfm?id=1855741.1855756

[23] A. Coen-Porisini, G. Denaro, C. Ghezzi, M. Pezzé, Using symbolic execution for verifying safety-critical systems, SIGSOFT Software Engineering Notes 26 (5) (2001) 142–151.
URL http://doi.acm.org/10.1145/503271.503230

[24] J. Jaffar, V. Murali, J. A. Navas, A. E. Santosa, TRACER: A symbolic execution tool for verification, in: Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings, Vol. 7358 of Lecture Notes in Computer Science, Springer, 2012, pp. 758–766. doi:10.1007/978-3-642-31424-7.

[25] D. A. Ramos, D. R. Engler, Practical, low-effort equivalence verification of real code, in: Proceedings of the 23rd international conference on Computer aided verification, CAV'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 669–685.
URL http://dl.acm.org/citation.cfm?id=2032305.2032360

[26] X. Leroy, Formal verification of a realistic compiler, Commun. ACM 52 (7) (2009) 107–115.
URL http://doi.acm.org/10.1145/1538788.1538814

[27] G. C. Necula, Translation validation for an optimizing compiler, in: M. S. Lam (Ed.), PLDI, ACM, 2000, pp. 83–94.

[28] A. M. Pitts, Operational semantics and program equivalence, in: Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures, Springer-Verlag, London, UK, UK, 2002, pp. 378–412.
URL http://dl.acm.org/citation.cfm?id=647424.725796

[29] T. Arons, E. Elster, L. Fix, S. Mador-Haim, M. Mishaeli, J. Shalev, E. Singerman, A. Tiemeyer, M. Y. Vardi, L. D. Zuck, Formal verification of backward compatibility of microcode, in: K. Etessami, S. K. Rajamani (Eds.), CAV, Vol. 3576 of Lecture Notes in Computer Science, Springer, 2005, pp. 185–198.

[30] S. Craciunescu, Proving the equivalence of clp programs, in: P. J. Stuckey (Ed.), ICLP, Vol. 2401 of Lecture Notes in Computer Science, Springer, 2002, pp. 287–301.

[31] O. Strichman, Regression verification: Proving the equivalence of similar programs, in: A. Bouajjani, O. Maler (Eds.), CAV, Vol. 5643 of Lecture Notes in Computer Science, Springer, 2009, p. 63.

[32] B. Godlin, O. Strichman, Inference rules for proving the equivalence of recursive procedures, in: Z. Manna, D. Peled (Eds.), Essays in Memory of Amir Pnueli, Vol. 6200 of Lecture Notes in Computer Science, Springer, 2010, pp. 167–184.

[33] K. Bae, S. Escobar, J. Meseguer, Abstract logical model checking of infinite-state systems using narrowing, in: F. van Raamsdonk (Ed.), 24th International Conference on Rewriting Techniques and Applications, RTA 2013, June 24-26, 2013, Eindhoven, The Netherlands, Vol. 21 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013, pp. 81–96. doi:10.4230/LIPIcs.RTA.2013.81.
URL http://dx.doi.org/10.4230/LIPIcs.RTA.2013.81

[34] C. Rocha, J. Meseguer, C. A. Muñoz, Rewriting modulo SMT and open system analysis, in: Rewriting Logic and Its Applications - 10th International Workshop, WRLA 2014, Held as a Satellite Event of ETAPS, Grenoble, France, April 5-6, 2014, Revised Selected Papers, 2014, pp. 247–262.

[35] C. Rocha, J. Meseguer, C. Muñoz, Rewriting modulo {SMT} and open system analysis, Journal of Logical and Algebraic Methods in Programming (2016) –doi:http://dx.doi.org/10.1016/j.jlamp.2016.10.001.
URL http://www.sciencedirect.com/science/article/pii/S2352220816301195

[36] A. Arusoaie, D. Lucanu, V. Rusu, A generic framework for symbolic execution, in: 6th International Conference on Software Language Engineering, Vol. 8225 of LNCS, Springer Verlag, 2013, pp. 281–301, also available as a technical report http://hal.inria.fr/hal-00853588.

[37] V. Rusu, A. Arusoaie, Proving reachability-logic formulas incrementally, in: D. Lucanu (Ed.), Rewriting Logic and Its Applications - 11th International Workshop, WRLA 2016, Held as a Satellite Event of ETAPS, Eindhoven, The Netherlands, April 2-3, 2016, Revised Selected Papers, Vol. 9942 of Lecture Notes in Computer Science, Springer, 2016, pp. 134–151.
URL http://dx.doi.org/10.1007/978-3-319-44802-2_8

[38] G. Roşu, C. Ellison, W. Schulte, Matching logic: An alternative to Hoare/Floyd logic, in: M. Johnson, D. Pavlovic (Eds.), Proceedings of the 13th International Conference on Algebraic Methodology And Software Technology (AMAST '10), Vol. 6486 of Lecture Notes in Computer Science, 2010, pp. 142–162.

[39] G. Roşu, A. Ştefănescu, Matching Logic: A New Program Verification Approach (NIER Track), in: ICSE'11: Proceedings of the 30th International Conference on Software Engineering, ACM, 2011, pp. 868–871.

[40] G. Roşu, Matching logic — extended abstract, in: Proceedings of the 26th International Conference on Rewriting Techniques and Applications (RTA'15), Vol. 36 of Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2015, pp. 5–21.

Appendix A. Proofs from Section 3

This section is dedicated to proving Lemmas 1 and 2 regarding the simulations between the concrete and symbolic transition relations. Proving Lemma 1 is actually quite a challenge. The proof includes several steps:

1. establishing a general result about unification by matching;
2. establishing a mutual simulation between $\Rightarrow_{\mathcal{S}^{\mathfrak{s}}}$ and another relation $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$, in which the details about rewriting with $\Rightarrow_{\mathcal{S}^{\mathfrak{s}}}$ are spelled out;
3. proving the simulation between the relations $\Rightarrow_{\mathcal{S}}$ and $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$ in two steps (and using the above-mentioned result on unification by matching)
   (a) when the relations are defined by RL formulas without quantifiers;
   (b) when the relations are defined by RL formulas with quantifiers.

Thus $\Rightarrow_{\mathcal{S}^{\mathfrak{s}}}$ simulates $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$, which simulates $\Rightarrow_{\mathcal{S}}$. By transitivity, $\Rightarrow_{\mathcal{S}^{\mathfrak{s}}}$ simulates $\Rightarrow_{\mathcal{S}}$, which is what Lemma 1 says (rule-by-rule). On the other hand, proving Lemma 2 is relatively easy. Its proof is shown at the end of this section.

Unification by matching. This result (Lemma 6 below) shows that, under certain conditions (Assumption 2), unification can be performed by matching.

Definition 14. *For a valuation $\rho : Var \to \mathcal{M}$ and a substitution $\sigma : X \to T_\Sigma(Y)$, we write $\rho \prec \sigma$ if $\rho|_X = (\rho \circ \sigma)|_X$.*

Lemma 5. *Let $\sigma_1 : X \to T_\Sigma(Y)$ and $\sigma_2 : Z \to T_\Sigma(X)$. If $\rho \prec \sigma_1$ and $\rho \prec \sigma_2$ then $\rho \prec \sigma_1 \circ \sigma_2$.*

Proof. Let $z \in Z$ be chosen arbitrarily. We have to show that $\rho(z) = \rho(\sigma_1(\sigma_2(z)))$.

First, $t_2 \triangleq \sigma_2(z)$ is a term over $T_\Sigma(X)$, and $t_1 \triangleq \sigma_1(\sigma_2(z))$ is a term over $T_\Sigma(Y)$, obtained by replacing in t_2 each $x \in \mathit{FreeVars}(t_2)$ by $\sigma_1(x) \in T_\Sigma(Y)$.

From $\rho \prec \sigma_1$ we know that $\rho(x) = \rho(\sigma_1(x))$ for all $x \in \mathit{FreeVars}(t_2)$. Thus, $\rho(t_2)$, obtained by replacing in t_2 function symbols f by their ρ-interpretation f_ρ and variables in $\mathit{FreeVars}(t_2)$ by their ρ-valuation $\rho(x)$, is also equal to the value obtained by replacing function symbols by their ρ-interpretation *and variables* $x \in \mathit{FreeVars}(t_2)$ *by* $\rho(\sigma_1(x))$. But the latter value is exactly $\rho(\sigma_1(t_2)) = \rho(\sigma_1(\sigma_2(z))$.

Thus, $\rho(t_2) = \rho(\sigma_1(\sigma_2(z))$, and from $\rho \prec \sigma_2$ we obtain $\rho(z) = \rho(\sigma_2(z)) = \rho(t_2)$. The conclusion follows by transitivity of equality. □

Remark 4. *The domains and codomains of substitutions σ_1, σ_2 in Lemma 5 were chosen so that the composition $\sigma_1 \circ \sigma_2 : Z \to T_\Sigma(Y)$ is well defined. The lemma can be generalized for substitutions $\sigma_1 : X \to T_\Sigma(Y)$ and $\sigma_2 : Z \to T_\Sigma(X')$ with $X \neq X'$ as follows: consider the substitution σ_1' with a domain $X \cup X'$, which is σ_1 extended as the identity on variables in $X' \setminus X$. By slight notation abuse we can define $\sigma_1 \circ \sigma_2$ as the (properly defined) composition $\sigma_1' \circ \sigma_2$. Now, from $\rho \prec \sigma_1$ we trivially obtain $\rho \prec \sigma_1'$, and, together with the hypothesis $\rho \prec \sigma_2$, Lemma 5 gives us $\rho \prec \sigma_1' \circ \sigma_2$, i.e., $\rho \prec \sigma_1 \circ \sigma_2$ since we defined $\sigma_1 \circ \sigma_2 \triangleq \sigma_1' \circ \sigma_2$. So, with the above generalized definition for the composition of substitutions, Lemma 5 holds.*

Lemma 6 (Unification by Matching)**.** *For all non-variable terms $t \in T_\Sigma(Var^b)$, linear terms $t' \in T_{\Sigma \setminus \Sigma^b}(Var)$, and all valuations $\rho : Var \to \mathcal{M}$ such that $\rho(t) = \rho(t')$, there exists a substitution σ such that $t =_A \sigma(t')$ and $\rho \prec \sigma$.*

Proof. Let $\mathcal{D}$ be the initial model of the builtin subtheory Σ^b. Let A be the set of non-builtin axioms, which by assumption in the paper are known to be linear, regular and non-collapsing. Note first that the initial model $\mathcal{M}$ is isomorphic to $T_{(\Sigma \setminus \Sigma^b)(\mathcal{D})}/A$, i.e., equivalence classes modulo A of ground terms in which the only subterms of a builtin sort are in $\mathcal{D}$. This is because $\mathcal{D}$ is itself the set of equivalence classes of of ground terms over Σ^b with respect to the set B of axioms of Σ^b, thus, $T_{(\Sigma \setminus \Sigma^b)(\mathcal{D})}/A$ amounts to first quotienting T_Σ by B and then by A, i.e., quotienting T_Σ by $A \cup B$ since $A \cap B = \emptyset$ (as they apply to different terms altogether).

The equality $\rho(t) = \rho(t')$ implies that for all $\hat{t} \in \rho(t)$ there exists $\hat{t}' \in \rho(t')$ such that $\hat{t} =_A \hat{t}'$. We fix arbitrarily such terms $\hat{t}, \hat{t}' \in T_{(\Sigma\setminus\Sigma^b)(\mathcal{D})}$. From $\hat{t}' \in \rho(t')$ we get that $\hat{t}'$ is obtained from $t' \in T_{(\Sigma\setminus\Sigma^b)}(\mathit{Var})$ by substituting variables $x_1 \dots x_n \in \mathit{FreeVars}(t')$ with representatives $\widehat{\rho(x_1)} \dots \widehat{\rho(x_n)}$ chosen in, respectively, $\rho(x_1) \dots \rho(x_n)$, i.e., $t' = c[x_1 \dots x_n]$ and $\hat{t}' = c[\widehat{\rho(x_1)} \dots \widehat{\rho(x_n)}]$ for some context c. The relation $\hat{t} =_A \hat{t}'$ is obtained using a finite number of axioms in A.

1. First we consider the case when no axioms are applied: $\hat{t} = \hat{t}'$, i.e., the two terms are syntactically equal. We prove by induction on positions (strings over natural numbers $\mathbb{N}$) that (♠) *any position ω in t' is also a position in t, and if t'_ω is a non-variable position then so is t_ω and the top function symbols of t'_ω and t_ω coincide.*

   (a) in the base case ω is the empty string which is obviously a position of t. Since t, t' are not variables they have some top symbols f, respectively g. From $\hat{t} = \hat{t}'$ we get $f = g$ which proves the base case.

   (b) for the inductive step: let ω be a position in t' of length $k+1$. Thus, there is a position ω' of length k, where $t'_{\omega'}$ has the form $f(\tau_1, \dots \tau_q)$ where $q > 0$ and f is a non-builtin function symbol. By induction hypothesis the position ω' is also a position of t. Now, $t_{\omega'}$ cannot be a variable; otherwise the subterm $\hat{t}_{\omega'}$ in $\hat{t}$ would have a builtin sort, which cannot be equal to the non-builtin $\hat{t}'_{\omega'}$. Thus, $t_{\omega'}$ is of the form $g(\tau'_1, \dots \tau'_r)$ for some $r \geq 0$. Now, the syntactical equality $\hat{t} = \hat{t}'$ implies $\hat{t}_{\omega'} = \hat{t}'_\omega$, where $\hat{t}$ has top symbol g and $\hat{t}$ has top symbol f: we obtain $f = g$ and $r = q > 0$. In particular, the position ω of length $k+1$ is also a position of t. Thus, t_ω exists, and so does $\hat{t}_\omega$, and $\hat{t} = \hat{t}'$ implies $\hat{t}_\omega = \hat{t}'_\omega$. Using the same reasoning as in the base case we obtain that if t'_ω is not a variable then so is t_ω, and the top function symbols of t'_ω and t_ω coincide, which proves the inductive step and (♠).

   Since $t' = c[x_1, \dots x_n]$ we obtain that the positions $\omega(x_1), \dots \omega(x_n)$ of the variables $x_1, \dots x_n$ in t' are also positions in t. We build the substitution σ by mapping x_i to $t_{\omega(x_i)}$ for $i = 1, \dots, n$ – it is a substitution, since t' is linear – and we obtain $\sigma(t') = t$. Moreover, from $\hat{t}' = c[\widehat{\rho(x_1)} \dots \widehat{\rho(x_n)}]$ we obtain $\widehat{\rho(x_i)} = \hat{t}'_{\omega(x_i)} = \hat{t}_{\omega(x_i)} \in \rho(t_{\omega(x_i)})$ and then $\rho(x_i) = [\widehat{\rho(x_i)}]_A = [\hat{t}_{\omega(x_i)}]_A = \rho(t_{\omega(x_i)}) = (\rho(\sigma((x_i)) = (\rho \circ \sigma)(x_i)$ which proves $\rho \prec \sigma$.

2. Next, we consider the case in which one axiom, say, $u = v$ is involved exactly once in establishing $\hat{t} =_A \hat{t}'$. By our assumption of the axioms A, u and v are linear terms that only contain non-builtin operations, and have the

same set of free variables, say, $\{y_1, \ldots y_m\}$. We assume without restriction of generality that $y_i \notin vars(t, t')$ for $i = 1, \ldots, m$. Thus, there is a substitution $\mu : \{y_1, \ldots, y_m\} \to T_{(\Sigma \setminus \Sigma^b)(\mathcal{D})}$ and a common position ω of $\hat{t}$ and $\hat{t}'$ such that:

- the terms $\hat{t}, \hat{t}'$ are equal except for their subterms at position ω, which is expressed as $\hat{t}[w]_\omega = \hat{t}'[w]_\omega$, where w is a fresh variable.
- the axiom $u = v$ with substitution μ equates the subterms at position ω, i.e., $\mu(u) = \hat{t}_\omega$, $\mu(v) = \hat{t}'_\omega$ or, symmetrically, $\mu(u) = \hat{t}'_\omega$, $\mu(v) = \hat{t}_\omega$. Since these equalities are obtained from the other one by switching u and v, and both terms u, v have the same properties, we can assume the first case: $\mu(u) = \hat{t}_\omega$ and $\mu(v) = \hat{t}'_\omega$.

Using the equalities established above the proof proceeds as follows:

- the fact that $\hat{t}_\omega = \mu(u)$ and $\hat{t}_\omega \in T_{(\Sigma \setminus \Sigma^b)(\mathcal{D})}$, and the fact that u is linear and has no builtin subterms other than variables, allow us to apply the same reasoning as in Item 1, and obtain a substitution $\sigma_u : \mathit{FreeVars}(u) \to T_\Sigma(\mathit{FreeVars}(t_\omega))$ such that $\sigma_u(u) = t_\omega$. Next, we note that our lemma holds or not independently of the value of ρ in $\mathit{Var} \setminus \mathit{FreeVars}(t, t')$. Thus, we can assume $\rho(y) = \rho(\sigma_u(y))$ for all $y \in \mathit{FreeVars}(u)$, i.e., $\rho \prec \sigma_u$.
- using $\mu(v) = \hat{t}'_\omega$ and the fact that t'_ω is linear and $\mu(v) \in T_{(\Sigma \setminus \Sigma^b)(\mathcal{D})}$, we apply again the same reasoning as in Item 1 and obtain $\sigma_v : \mathit{FreeVars}(t'_\omega) \to T_\Sigma(\mathit{FreeVars}(v))$ such that $\sigma_v(t'_\omega) = v$ and $\rho \prec \sigma_v$ (by extending σ_v to the identity over $\mathit{FreeVars}(t_\omega) \setminus \mathit{FreeVars}(t'_\omega)$).
- from $\hat{t}[w]_\omega = \hat{t}'[w]_\omega$, by using once more the reasoning in Item 1, we obtain that each position ω' of $t'[w]_\omega$, ω' is also a position of $t[w]_\omega$ and if ω' is not a variable position then the top symbols of $(t'[w]_\omega)_{\omega'}$ and $(t[w]_\omega)_{\omega'}$ coincide. We construct a substitution σ' over $\{w\} \uplus \mathit{FreeVars}(t') \setminus \mathit{FreeVars}(t'_\omega)$ with $\sigma'(w) = w$, such that $\sigma'(t'[w]_\omega) = t[w]_\omega$ and $\rho \prec \sigma'$ (again, σ' is the identity outside its domain).
- $(\sigma_u \circ \sigma')(t'[u]_\omega) = \sigma_u(t[\sigma'(u)]_\omega) = \sigma_u(t[u]_\omega) = t[\sigma_u(u)]_\omega = t[t_\omega]_\omega = t$;
- $t'[u]_\omega =_A t'[v]_\omega = t'[\sigma_v(t'_\omega)]_\omega = \sigma_v(t')$, thus, $((\sigma_u \circ \sigma') \circ \sigma_v)(t') =_A t$.
- finally, let $\sigma \triangleq ((\sigma_u \circ \sigma') \circ \sigma_v)$; we have obtained $\sigma(t') =_A t$ above, and $\rho \prec \sigma$ follows from $\rho \prec \sigma_v$, $\rho \prec \sigma_u$, $\rho \prec \sigma'$ and Remark 4.

3. There remains to consider the case when more than one axiom is involved in the relation $\hat{t} =_A \hat{t}'$. Thus, there are axioms $a_1, \ldots, a_p$ $(p > 1)$ and terms in $T_{(\Sigma \setminus \Sigma^b)(\mathcal{D})}$: $\hat{t}_0 \triangleq \hat{t}, \hat{t}_1, \ldots, \hat{t}_p \triangleq \hat{t}'$ such that $\hat{t} \triangleq \hat{t}_0 =_{a_1} \hat{t}_1 \cdots =_{a_p} \hat{t}_p \triangleq \hat{t}'$.

Now, for each of the *ground* terms $\hat{t}_1, \ldots, \hat{t}_{p-1}$ in the sequence we construct a corresponding term *with variables* $t_1, \ldots, t_{p-1}$, such that t_i is obtained from $\hat{t}_i$ by substituting constants in $\mathcal{D}$ with *fresh* variables in Var^b. Thus, $t_1 \ldots t_{p-1}$ are linear and belong to $T_{(\Sigma \setminus \Sigma^b)}(Var^b)$ (since the corresponding ground terms $\hat{t}_i$ are in $T_{(\Sigma \setminus \Sigma^b)(\mathcal{D})}$). Remember also that $t' \in T_{(\Sigma \setminus \Sigma^b)}(Var)$ is linear by hypothesis. Thus, we can repeatedly apply the reasoning at Item 2 and obtain substitutions $\sigma_i : FreeVars(t_i) \to T_\Sigma(FreeVars(t_{i-1}))$ such that $\sigma_i(t_i) =_A t_{i-1}$ and $\rho \prec \sigma_i$, for $i = 1, \ldots, p$. With $\sigma = \sigma_p \circ \sigma_{p-1} \cdots \circ \sigma_1$ we get $\sigma(t') =_A t$ and (by Remark 4) $\rho \prec \sigma$. This concludes the proof.

□

A relation $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$ and its mutual simulation with $\Rightarrow_{\mathcal{S}^{\mathfrak{s}}}$. The following definition introduces the relation $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$ on patterns. It is a version of the rewriting-based relation $\Rightarrow_{\mathcal{S}^{\mathfrak{s}}}$ where details about rewriting (matching with a substitution, applying the substitution to the right-hand side of a rule) are spelled out.

Definition 15 (relation $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$)**.** *Let φ be a pattern with $\varphi \triangleq (\exists X)\pi \wedge \phi$ and $\alpha \triangleq \varphi_l \Rightarrow \varphi_r$ be an* RL *formula with $\varphi_l \triangleq \pi_l \wedge \phi_l$, $\varphi_r \triangleq (\exists Y)\pi_r \wedge \phi_r$, and $(X \cup FreeVars(\varphi)) \cap (Y \cup FreeVars(\varphi_l, \varphi_r)) = \emptyset$. We write $\varphi \Rightarrow^{\mathfrak{s}}_{\alpha} \varphi'$, with $\varphi' \triangleq (\exists X, Y)\pi' \wedge \phi'$, whenever there exists a matcher $\sigma \in match_{\cong}(\pi, \pi_l)$ such that $\pi' = \sigma'(\pi_r)$ and $\phi' = \phi \wedge \sigma'(\phi_l \wedge \phi_r)$ where $\sigma' = \sigma \cup Id|_{Var \setminus FreeVars(\pi_l)}$.*

The following lemma establishes a mutual simulation between $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$ and $\Rightarrow_{\mathcal{S}^{\mathfrak{s}}}$. Remember that patterns $(\exists X)\pi \wedge \phi$ can equivalently be seen as terms of sort $Cfg^{\mathfrak{s}}$, and that we extended the congruence $\cong$ from terms of sort Cfg to terms of sort $Cfg^{\mathfrak{s}}$, as follows: $(\exists X)\pi_1 \wedge \phi \cong (\exists X)\pi_2 \wedge \phi$ iff $\pi_1 \cong \pi_2$. For convenience we also recall here the definition of the relation $\Rightarrow_{\mathcal{S}^{\mathfrak{s}}}$ (Def. 7):

for $\alpha^{\mathfrak{s}} \triangleq (\exists \mathcal{X})\pi_l \wedge \psi \Rightarrow (\exists \mathcal{X}, Y)\pi_r \wedge (\psi \wedge \phi_l \wedge \phi_r) \in \mathcal{S}^{\mathfrak{s}}$ we write $(\exists X)\pi \wedge \phi \Rightarrow_{\alpha^{\mathfrak{s}}} (\exists X, Y)\pi' \wedge \phi'$ if $(\exists X)\pi \wedge \phi$ is rewritten by $\alpha^{\mathfrak{s}}$ to $(\exists X, Y)\pi' \wedge \phi'$, i.e., there is σ'' such that $\sigma''((\exists \mathcal{X})\pi_l \wedge \psi) \cong (\exists X)\pi \wedge \phi$ and $(\exists X, Y)\pi' \wedge \phi' = \sigma''((\exists \mathcal{X}, Y)\pi_r \wedge (\psi \wedge \phi_l \wedge \phi_r)$.

Lemma 7. *$(\exists X)\pi \wedge \phi \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} (\exists X, Y)\pi' \wedge \phi'$ iff $(\exists X)\pi \wedge \phi \Rightarrow_{\mathcal{S}^{\mathfrak{s}}} (\exists X, Y)\pi' \wedge \phi'$.*

Proof. ($\Rightarrow$) Assume $(\exists X)\pi \wedge \phi \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} (\exists X, Y)\pi' \wedge \phi'$, thus, there exists $\alpha \triangleq \pi_l \wedge \phi_l \Rightarrow (\exists Y)\pi_r \wedge \phi_r \in \mathcal{S}$, $\sigma : FreeVars(\pi_l) \to T_\Sigma(FreeVars(\pi)) \in match_{\cong}(\pi, \pi_l)$, and $\sigma' \triangleq \sigma \cup Id|_{Var \setminus FreeVars(\pi_l)}$ satisfies $\pi' = \sigma'(\pi_r)$ and $\phi' = \phi \wedge \sigma'(\phi_l \wedge \phi_r)$.

Consider $\alpha^{\mathfrak{s}} \triangleq (\exists \mathcal{X})\pi_l \wedge \psi \Rightarrow (\exists \mathcal{X}, Y)\pi_r \wedge (\psi \wedge \phi_l \wedge \phi_r) \in \mathcal{S}^{\mathfrak{s}}$ and the substitution $\sigma'' \triangleq \sigma' \cup (\mathcal{X} \leftarrow X) \cup (\psi \leftarrow \phi)$. We have: $\sigma''((\exists \mathcal{X})\pi_l \wedge \psi) \cong (\exists X)\pi \wedge \phi$; (remember how we extended the congruence $\cong$ to terms in $Cfg^{\mathfrak{s}}$ by letting $(\exists X)\pi_1 \wedge \phi \cong (\exists X)\pi_2 \wedge \phi$ if and only if $\pi_1 \cong \pi_2$); and we have $\sigma''((\exists \mathcal{X}, Y)\pi_r \wedge (\psi \wedge \phi_l \wedge \phi_r)) = (\exists X, Y)\sigma'(\pi_r) \wedge (\phi \wedge \sigma'(\phi_l \wedge \phi_r)) = (\exists X, Y)\pi' \wedge \phi'$.

Thus, $(\exists X)\pi\wedge\phi \Rightarrow_{\alpha^{\mathfrak{s}}} (\exists X, Y)\pi'\wedge\phi'$, which proves the ($\Rightarrow$) implication.

($\Leftarrow$) Assume $(\exists X)\pi\wedge\phi \Rightarrow_{\mathcal{S}^{\mathfrak{s}}} (\exists X, Y)\pi'\wedge\phi'$. Thus, there exist $\alpha^{\mathfrak{s}} \triangleq (\exists \mathcal{X})\pi_l\wedge\psi \Rightarrow (\exists \mathcal{X}, Y)\pi_r\wedge(\psi\wedge\phi_l\wedge\phi_r) \in \mathcal{S}^{\mathfrak{s}}$ and a substitution σ'' from $\mathit{FreeVars}((\exists \mathcal{X})\pi_l\wedge\psi)$ to terms over $\mathit{FreeVars}((\exists X)\pi\wedge\phi)$, such that

1. $\sigma''((\exists \mathcal{X})\pi_l\wedge\psi) \cong (\exists X)\pi\wedge\phi$, where the congruence $\cong$ over $\mathit{Cfg}^{\mathfrak{s}}$ satisfies $(\exists X)\pi_1\wedge\phi \cong (\exists X)\pi_2\wedge\phi$ if and only if $\pi_1 \cong \pi_2$. *Due to this property,* σ'' has the form $\sigma' \cup (\mathcal{X} \leftarrow X) \cup (\psi \leftarrow \phi)$ with $\sigma' = \sigma \cup \mathit{Id}|_{\mathit{Var}\setminus\mathit{FreeVars}(\pi_l)}$ and $\sigma : \mathit{FreeVars}(\pi_l) \rightarrow T_\Sigma(\mathit{FreeVars}(\pi)) \in \mathit{match}_{\cong}(\pi, \pi_l)$. We obtain $\sigma'(\pi_l) \cong \pi$, $\sigma''(\psi) = \phi$, and $\sigma''(\mathcal{X}) = X$;
2. $\sigma''((\exists \mathcal{X}, Y)\pi_r\wedge(\psi\wedge\phi_l\wedge\phi_r)) = (\exists X, Y)\pi'\wedge\phi'$, thus, we obtain $\sigma''(\pi_r) = \sigma'(\pi_r) = \pi'$ and also $\sigma''(\psi\wedge\phi_l\wedge\phi_r) = \phi\wedge\sigma'(\phi_l\wedge\phi_r) = \phi'$

We have thus obtained $\pi' = \sigma'(\pi_r)$ and $\phi' = \phi\wedge\sigma'(\phi_l\wedge\phi_r)$ for some σ' s.t. $\sigma'(\pi_l) \cong \pi$, which implies $(\exists X)\pi\wedge\phi \Rightarrow^{\mathfrak{s}}_{\alpha} (\exists X, Y)\pi'\wedge\phi'$. This proves the ($\Leftarrow$) implication and concludes the lemma. □

Simulation of $\Rightarrow_{\mathcal{S}}$ by $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$. We have established mutual simulation of $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$ and $\Rightarrow_{\mathcal{S}^{\mathfrak{s}}}$; there remains to prove the simulation of $\Rightarrow_{\mathcal{S}}$ by $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$. We will do this in two steps. In the first step we shall consider relations $\Rightarrow_{\mathcal{S}}$ and $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$ defined by RL formulas $\varphi \Rightarrow \varphi'$ in which existential quantifiers in the right-hand side have been replaced by fresh variables. Thus, we have formulas satisfying $\mathit{FreeVars}(\varphi') \not\subseteq \mathit{FreeVars}(\varphi)$, for which the definition of validity becomes:

Definition 16. *An* RL *formula $\varphi \Rightarrow \varphi'$ with $\mathit{FreeVars}(\varphi') \not\subseteq \mathit{FreeVars}(\varphi)$ is valid w.r.t. a set $\mathcal{S}$ of* RL *formulas, written $\mathcal{S} \models \varphi \Rightarrow \varphi'$, if for all (γ_0, ρ) such that $(\gamma_0, \rho) \models \varphi$, and all complete paths $\gamma_0 \Rightarrow_{\mathcal{S}} \cdots \Rightarrow_{\mathcal{S}} \gamma_n$, there are $0 \leq i \leq n$ and a valuation ρ' with $\rho'|_{\mathit{FreeVars}(\varphi)} = \rho|_{\mathit{FreeVars}(\varphi)}$ such that $(\gamma_i, \rho') \models \varphi'$.*

Remark 5. *The new valuation ρ' is required in order to avoid the undesired capturing of additional variables (in $\mathit{FreeVars}(\varphi') \setminus \mathit{FreeVars}(\varphi)$) by the valuation ρ; however, for variables of φ the two valuations coincide.*

We shall also use the following remarks, which follow from definitions.

Remark 6. *If $\mathcal{S} \models \alpha$ with $\alpha \triangleq \varphi \Rightarrow \varphi'$ then for all pairs (γ_0, ρ) such that $(\gamma_0, \rho) \models \varphi$, all complete paths $\gamma_0 \Rightarrow_{\mathcal{S}} \cdots \Rightarrow_{\mathcal{S}} \gamma_n$, and all $i \in \{0, \ldots, n\}$ such that $(\gamma_i, \rho') \models \varphi'$ for some ρ' with $\rho'|_{\mathit{FreeVars}(\varphi)} = \rho|_{\mathit{FreeVars}(\varphi)}$: $\gamma_0 \Rightarrow_{\alpha} \gamma_i$.*

Remark 7. *From Assumptions 1 and 3 it follows that for all rules $\pi_l\wedge\phi_l \Rightarrow (\exists Y)\pi_r\wedge\phi_r$, $\mathit{FreeVars}(\pi_r\wedge\phi_r) \setminus \mathit{FreeVars}(\pi_l\wedge\phi_l) \subseteq Y \subseteq \mathit{Var}^b$.*

The next remark regards the definition of the "restricted" relation $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$, i.e., generated from unquantified RL formulas with additional variables in their right-hand side as discussed above. It instantiates Def. 15 to this case.

Remark 8 (restricted relation $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$)**.** *Let φ be a pattern with $\varphi \triangleq \pi \wedge \phi$ and $\alpha \triangleq \varphi_l \Rightarrow \varphi_r$ be an* RL *formula with $\varphi_l \triangleq \pi_l \wedge \phi_l$, $\varphi_r \triangleq \pi_r \wedge \phi_r$, and $FreeVars(\varphi) \cap FreeVars(\varphi_l, \varphi_r) = \emptyset$. We have $\varphi \Rightarrow^{\mathfrak{s}}_{\alpha} \varphi'$, with $\varphi' \triangleq \pi' \wedge \phi'$, whenever there exists a matcher $\sigma \in match_{\cong}(\pi, \pi_l)$ such that $\pi' = \sigma'(\pi_r)$ and $\phi' = \phi \wedge \sigma'(\phi_l \wedge \phi_r)$ where $\sigma' = \sigma \cup Id|_{Var \setminus FreeVars(\pi_l)}$.*

Lemma 8 (restricted $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$ simulates $\Rightarrow_{\mathcal{S}}$)**.** *For all $\gamma, \gamma' \in \mathcal{M}_{Cfg}$, pattern $\varphi \triangleq \pi \wedge \phi$ with $FreeVars(\varphi) \subseteq Var^b$, and valuation ρ, if $(\gamma, \rho) \models \varphi$ and $\gamma \Rightarrow_{\alpha} \gamma'$ then there is $\varphi' \triangleq \pi' \wedge \phi'$ with $FreeVars(\varphi') \subseteq Var^b$ such that $\varphi \Rightarrow^{\mathfrak{s}}_{\alpha} \varphi'$ and $(\gamma', \rho') \models \varphi'$, for some valuation ρ' such that $\rho'|_{FreeVars(\varphi)} = \rho|_{FreeVars(\varphi)}$.*

Proof. From $\gamma \Rightarrow_{\alpha} \gamma'$ we obtain $\alpha \triangleq \varphi_l \Rightarrow \varphi_r \in \mathcal{S}$ and $\varphi_l \triangleq \pi_l \wedge \phi_l$, $\varphi_r \triangleq \pi_r \wedge \phi_r$, and a valuation μ such that $(\gamma, \mu) \models \pi_l \wedge \phi_l$ and $(\gamma', \mu) \models \pi_r \wedge \phi_r$. Since the rules are defined up to the names of their free variables, we can assume $FreeVars(\varphi) \cap FreeVars(\varphi_l, \varphi_r) = \emptyset$. Let then ρ'' be any valuation such that $\rho''|_{FreeVars(\varphi)} = \rho|_{FreeVars(\varphi)}$, $\rho''|_{FreeVars(\varphi_l, \varphi_r)} = \mu|_{FreeVars(\varphi_l, \varphi_r)}$. We thus have

1. $(\gamma, \rho'') \models \pi \wedge \phi$, hence, (i) $\gamma = \rho''(\pi)$ and (iv) $\rho'' \models \phi$;
2. $(\gamma, \rho'') \models \pi_l \wedge \phi_l$, hence, (ii) $\gamma = \rho''(\pi_l)$ and (v) $\rho'' \models \phi_l$;
3. $(\gamma', \rho'') \models \pi_r \wedge \phi_r$, hence, (iii) $\gamma' = \rho''(\pi_r)$ and (vi) $\rho'' \models \phi_r$.

From (i) and (ii) we obtain $\rho''(\pi) = \rho''(\pi_l)$ and, using Lemma 6 (unification by matching) we obtain $\sigma : FreeVars(\pi_l) \to T_{\Sigma}(FreeVars(\pi)) \in match_{\cong}(\pi, \pi_l)$ such that $\rho'' \prec \sigma$, that is, $\rho''|_{FreeVars(\pi_l)} = (\rho'' \circ \sigma)|_{FreeVars(\pi_l)}$. Let $\sigma' \triangleq \sigma \cup Id|_{Var \setminus FreeVars(\pi_l)}$. We have (vii) $\rho'' = \rho'' \circ \sigma'$.

Let $\pi' \triangleq \sigma'(\pi_r)$, $\phi' \triangleq \phi \wedge \sigma'(\phi_l \wedge \phi_r)$, $\varphi' \triangleq \pi' \wedge \phi'$. Using Remark 8 we obtain $\varphi \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \varphi'$. Moreover, $FreeVars(\varphi') \subseteq Var^b$ since $FreeVars(\varphi') = FreeVars(\sigma'(\pi_r)) \cup FreeVars(\phi) \cup FreeVars(\sigma'(\phi_l)) \cup FreeVars(\sigma'(\phi_r))$, and σ' maps $FreeVars(\pi_l)$ to terms over $FreeVars(\pi) \subseteq Var^b$ and each of the sets $FreeVars(\pi_r) \setminus FreeVars(\pi_l)$, $FreeVars(\phi_r) \setminus FreeVars(\pi_l)$, which are subsets of $FreeVars(\varphi_r) \setminus FreeVars(\varphi_l) \subseteq Var^b$, to the identity. Note that Remark 7 was used in the above reasoning.

There remains to find ρ' with $\rho'|_{FreeVars(\varphi)} = \rho|_{FreeVars(\varphi)}$ such that $(\gamma', \rho') \models \varphi'$. We let $\rho' \triangleq \rho''$, which satisfies $\rho''|_{FreeVars(\varphi)} = \rho|_{FreeVars(\varphi)}$ by construction.

1. From (iii) we have $\gamma' = \rho''(\pi_r)$, and using (vii), $\boxed{\gamma' = \rho''(\sigma'(\pi_r))}$;
2. from (iv) : $\boxed{\rho'' \models \phi}$;

3. from (v), $\rho'(\phi_l) = \mathit{true}$. Using (vii), $\rho'((\sigma'(\phi_l)) = \mathit{true}$, i.e., $\boxed{\rho'' \models \sigma'(\phi_l)}$;

4. from (vi), $\rho'(\phi_r) = \mathit{true}$. Using (vii), $\rho'((\sigma'(\phi_r)) = \mathit{true}$, i.e., $\boxed{\rho'' \models \sigma'(\phi_r)}$.

The boxed conclusions of items 1-4 imply $(\gamma', \rho'') \models \sigma'(\pi_r) \wedge \phi \wedge \sigma'(\varphi_l \wedge \varphi_r)$, i.e., $(\gamma', \rho'') \models \varphi'$, which concludes the proof of the lemma. □

There remains to prove that the general relation $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$ simulates $\Rightarrow_{\mathcal{S}}$.

Lemma 9 (general $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$ simulates $\Rightarrow_{\mathcal{S}}$)**.** *For all $\gamma, \gamma' \in \mathcal{M}_{Cfg}$, pattern $\varphi \triangleq (\exists X)\pi \wedge \phi$ with $X \cup \mathit{FreeVars}(\varphi) \subseteq \mathit{Var}^b$, and valuation ρ, if $(\gamma, \rho) \models \varphi$ and $\gamma \Rightarrow_\alpha \gamma'$ with $\alpha \triangleq \varphi_l \Rightarrow \varphi_r$, $\varphi_l \triangleq \pi_l \wedge \phi_l$, $\varphi_r \triangleq (\exists Y)\pi_r \wedge \phi_r$, then there is $\varphi' \triangleq (\exists X, Y)\pi' \wedge \phi'$ with $X \cup Y \cup \mathit{FreeVars}(\varphi') \subseteq \mathit{Var}^b$ such that $\varphi \Rightarrow^{\mathfrak{s}}_\alpha \varphi'$ and $(\gamma', \rho) \models \varphi'$.*

Proof. Consider the unquantified pattern $\pi \wedge \phi$ and the unquantified rule $\alpha' \triangleq \pi_l \wedge \phi_l \Rightarrow \pi_r \wedge \phi_r$. We have $\mathit{FreeVars}(\pi \wedge \phi) = X \cup \mathit{FreeVars}(\varphi) \subseteq \mathit{Var}^b$ using our lemma's hypotheses. We also have the hypothesis $\gamma \Rightarrow_\alpha \gamma'$, thus, there exist valuations μ, μ' such that $\mu'|_{\mathit{Var} \setminus Y} = \mu|_{\mathit{Var} \setminus Y}$, $(\gamma, \mu) \models \pi_l \wedge \phi_l$, and $(\gamma', \mu') \models \pi_r \wedge \phi_r$. Since $\mathit{FreeVars}(\pi_l \wedge \phi_l) \subseteq (\mathit{Var} \setminus Y)$ we also have $(\gamma, \mu') \models \pi_l \wedge \phi_l$. Thus, using the unquantified version of α, i.e., $\alpha' \triangleq \pi_l \wedge \phi_l \Rightarrow \pi_r \wedge \phi_r$, we obtain $\gamma \Rightarrow_{\alpha'} \gamma'$.

We can now apply Lemma 8 and obtain a pattern $\pi' \wedge \phi'$ with $\mathit{FreeVars}(\pi' \wedge \phi') \subseteq \mathit{Var}^b$ such that $\pi \wedge \phi \Rightarrow^{\mathfrak{s}}_{\alpha'} \pi' \wedge \phi'$ and $(\gamma', \rho') \models \pi' \wedge \phi'$, for some valuation ρ' such that $\rho'|_{\mathit{FreeVars}(\pi \wedge \phi)} = \rho|_{\mathit{FreeVars}(\pi \wedge \phi)}$.

Specifically, from the proof of Lemma 7 we have $\pi' \triangleq \sigma'(\pi_r)$, $\phi' \triangleq \phi \wedge \sigma'(\phi_l \wedge \phi_r)$, with $\sigma' \triangleq \sigma \cup \mathit{Id}|_{\mathit{Var} \setminus \mathit{FreeVars}(\pi_l)}$ and $\sigma : \mathit{FreeVars}(\pi_l) \to T_\Sigma(\mathit{FreeVars}(\pi)) \in \mathit{match}_{\cong}(\pi, \pi_l)$. Let $\varphi' \triangleq (\exists X, Y)(\sigma'(\pi_r) \wedge \phi \wedge \sigma'(\phi_l \wedge \phi_r)$. By Def. 15, $\varphi \Rightarrow^{\mathfrak{s}}_\alpha \varphi'$.

We first show $\mathit{FreeVars}(\varphi') \subseteq \mathit{FreeVars}(\varphi)$. We have $\varphi \triangleq (\exists X)\pi \wedge \phi$ and, since $\varphi' = (\exists X, Y)(\sigma'(\pi_r) \wedge \phi \wedge \sigma'(\phi_l \wedge \phi_r))$, $\mathit{FreeVars}(\varphi') = (\mathit{FreeVars}(\sigma'(\pi_r)) \cup \mathit{FreeVars}(\phi) \cup \mathit{FreeVars}(\sigma'(\phi_l)) \cup \mathit{FreeVars}(\sigma'(\phi_r))) \setminus (X \cup Y)$.

We have $\mathit{FreeVars}(\sigma'(\pi_r)) = \mathit{FreeVars}(\pi) \cup (\mathit{FreeVars}(\pi_r) \setminus \mathit{FreeVars}(\pi_l))$, and then $\mathit{FreeVars}(\sigma'(\pi_r)) \setminus (X \cup Y) = (\mathit{FreeVars}(\pi) \setminus X) \cup ((\mathit{FreeVars}(\pi_r) \setminus Y) \setminus \mathit{FreeVars}(\pi_l))$. But $(\mathit{FreeVars}(\pi) \setminus X) \subseteq \mathit{FreeVars}(\varphi)$ and by Assumption 1, $(\mathit{FreeVars}(\pi_r) \setminus Y) \setminus \mathit{FreeVars}(\pi_l) = \emptyset$. Hence, $\mathit{FreeVars}(\sigma'(\pi_r)) \subseteq \mathit{FreeVars}(\varphi)$.

Then, $\mathit{FreeVars}(\phi) \setminus (X \cup Y) = \mathit{FreeVars}(\phi) \setminus X \subseteq \mathit{FreeVars}(\varphi)$.

Next, $\mathit{FreeVars}(\sigma'(\phi_l)) \subseteq \mathit{FreeVars}(\sigma'(\pi_l))$ using Assumption 1, and we obtain $\mathit{FreeVars}(\sigma'(\phi_l)) \subseteq \mathit{FreeVars}(\pi)$, and then $\mathit{FreeVars}(\sigma'(\phi_l)) \setminus (X \cup Y) = \mathit{FreeVars}(\sigma'(\phi_l)) \setminus X \subseteq \mathit{FreeVars}(\pi) \setminus X \subseteq \mathit{FreeVars}(\varphi)$.

Finally, by Assumption 1 we have $\mathit{FreeVars}(\phi_r) \subseteq \mathit{FreeVars}(\pi_l) \cup Y$, thus, $\mathit{FreeVars}(\sigma'(\phi_r)) \subseteq \mathit{FreeVars}(\pi) \cup Y$. We then obtain $\mathit{FreeVars}(\sigma'(\phi_r)) \setminus (X \cup Y) \subseteq \mathit{FreeVars}(\sigma'(\phi_r)) \setminus Y \subseteq \mathit{FreeVars}(\pi) \subseteq \mathit{FreeVars}(\varphi)$.

The proof of $\mathit{FreeVars}(\varphi') \subseteq \mathit{FreeVars}(\varphi)$ is now complete.

Since we already obtained $\varphi \Rightarrow^{\mathfrak{s}}_{\alpha} \varphi'$ there only remains to prove $(\gamma', \rho) \models \varphi'$ and $X \cup Y \cup \mathit{FreeVars}(\varphi') \subseteq Var^b$.

We now prove $(\gamma', \rho) \models \varphi'$. Using Lemma 8 we obtained above $(\gamma', \rho') \models \pi' \wedge \phi'$, for some valuation ρ' such that $\rho'|_{\mathit{FreeVars}(\pi \wedge \phi)} = \rho|_{\mathit{FreeVars}(\pi \wedge \phi)}$. Thus, using the definition of valuation of quantified patterns, we also obtain $(\gamma', \rho') \models (\exists X, Y)\pi' \wedge \phi'$, i.e., $(\gamma', \rho') \models \varphi'$. From $\rho'|_{\mathit{FreeVars}(\pi \wedge \phi)} = \rho|_{\mathit{FreeVars}(\pi \wedge \phi)}$ we also have $\rho'|_{\mathit{FreeVars}(\pi \wedge \phi) \setminus X} = \rho|_{\mathit{FreeVars}(\pi \wedge \phi) \setminus X}$, that is $\rho'|_{\mathit{FreeVars}(\varphi)} = \rho|_{\mathit{FreeVars}(\varphi)}$. Using $\mathit{FreeVars}(\varphi') \subseteq \mathit{FreeVars}(\varphi)$ (established at the beginning of this proof) we obtain $\rho'|_{\mathit{FreeVars}(\varphi')} = \rho|_{\mathit{FreeVars}(\varphi')}$, which together with $(\gamma', \rho') \models \varphi'$ proves $(\gamma', \rho) \models \varphi'$.

There only remains to be proved that $X \cup Y \cup \mathit{FreeVars}(\varphi') \subseteq Var^b$. We have $X \cup Y \cup \mathit{FreeVars}(\varphi') = \mathit{FreeVars}(\pi' \wedge \phi')$ and by Lemma 8 we have that $\mathit{FreeVars}(\pi' \wedge \phi') \subseteq Var^b$. This concludes the proof of our lemma. □

Now, Lemma 1 is a corollary of Lemmas 9 and 7.

Simulation of $\Leftarrow_{\mathcal{S}^{\mathfrak{s}}}$ by $\Leftarrow_{\mathcal{S}}$. Lemma 2 is the last result in Section 3.

Lemma 2($\Leftarrow_{\alpha}$ simulates $\Leftarrow_{\alpha^{\mathfrak{s}}}$). *For all all patterns φ, φ' such that $\varphi \Rightarrow_{\alpha^{\mathfrak{s}}} \varphi'$, for all configurations $\gamma' \in \mathcal{M}_{Cfg}$ and valuations ρ such that $(\gamma', \rho) \models \varphi'$, there exists a configuration $\gamma \in \mathcal{M}_{Cfg}$ such that $(\gamma, \rho) \models \varphi$ and $\gamma \Rightarrow_{\alpha} \gamma'$.*

Proof. Let $\varphi \triangleq (\exists X)\pi \wedge \phi$ and $\alpha \triangleq \pi_l \wedge \phi_l \Rightarrow (\exists Y)\pi_r \wedge \phi_r$. Thus we have $\varphi' = (\exists X, Y)\sigma'(\pi_r) \wedge (\phi \wedge \sigma'(\phi_r) \wedge \sigma'(\phi_l))$ for some substitution $\sigma' : \mathit{FreeVars}(\pi_l) \rightarrow T_{\Sigma}(\mathit{FreeVars}(\pi))$ (extended to the identity over $\mathit{Var} \setminus \mathit{FreeVars}(\pi_l)$) such that $\pi \cong \sigma'(\pi_l)$. From $(\gamma', \rho) \models \varphi'$ we obtain that there is ρ' with $\rho'|_{\mathit{Var} \setminus (X \cup Y)} = \rho|_{\mathit{Var} \setminus (X \cup Y)}$ such that $\gamma' = \rho'(\sigma'(\pi_r))$ and $\rho' \models (\phi \wedge \sigma'(\phi_r) \wedge \sigma'(\phi_l))$. Let $\gamma \triangleq \rho'(\pi)$. We thus have $(\gamma, \rho') \models \pi \wedge \phi$ and then $(\gamma, \rho) \models (\exists X)\pi \wedge \phi = \varphi$.

There only remains to prove that $\gamma \Rightarrow_{\alpha} \gamma'$. From $\gamma = \rho'(\pi)$ and $\pi \cong \sigma'(\pi_l)$ we obtain $\gamma = (\rho' \circ \sigma')(\pi_l)$. From $\rho' \models (\phi \wedge \sigma'(\phi_r) \wedge \sigma'(\phi_l))$ we obtain $(\rho' \circ \sigma') \models \phi_l$ and thus $(\gamma, \rho' \circ \sigma') \models \pi_l \wedge \phi_l$. From $\gamma' = \rho'(\sigma'(\pi_r))$ and $\rho' \models \sigma'(\phi_r)$ we obtain $(\gamma', \rho' \circ \sigma') \models \pi_r \wedge \phi_r$ which implies $(\gamma', \rho' \circ \sigma') \models (\exists Y)\pi_r \wedge \phi_r$.

Finally, $(\gamma, \rho' \circ \sigma') \models \pi_l \wedge \phi_l$ and $(\gamma', \rho' \circ \sigma') \models (\exists Y)\pi_r \wedge \phi_r$ and $\alpha \triangleq \pi_l \wedge \phi_l \Rightarrow (\exists Y)\pi_r \wedge \phi_r$ together mean $\gamma \Rightarrow_{\alpha} \gamma'$, which concludes the proof. □

Appendix B. Proofs from Section 4

We shall use the following notation: $\mathcal{S}_0 \models_n \mathcal{G}$ whenever for all pairs (γ_0, ρ) such that $(\gamma_0, \rho) \models \varphi$, and all complete paths $\gamma_0 \Rightarrow_{\mathcal{S}} \cdots \Rightarrow_{\mathcal{S}} \gamma_n$ *of length at most* n, there exists $0 \leq i \leq n$ such that $(\gamma_i, \rho) \models \varphi'$.

Lemma 10 (Simulation by Graph)**.** *Consider any complete path* $\tau = \gamma_0 \Rightarrow_{\alpha_1} \cdots \Rightarrow_{\alpha_n} \gamma_n$ *with* $\alpha_1, \ldots, \alpha_n \in \mathcal{S}$*, s.t.* $(\gamma_0, \rho) \models \pi \wedge \phi$*. Assume* $\mathcal{S} \models_{n-1} \mathcal{G}$*. Then, there exist:* $k \geq 0$*, a subsequence* $(0 = i_0 < \cdots < i_k = n)$*, and a path* $\pi \wedge \phi = \varphi_0 \xrightarrow{\alpha_{i_1}} \cdots \xrightarrow{\alpha_{i_k}} \varphi_n$ *in the graph constructed by the procedure in Fig. 7 such that* $(\gamma_{i_j}, \rho) \models \varphi_{i_j}$ *for* $j = 0 \ldots k$*.*

Proof. We show how to inductively construct the sequence of indices $(0 = i_0 < \cdots < i_k = n)$ and the corresponding path in the graph.

The first index is (by definition) $i_0 = 0$. In this case the path in the graph reduces to the sole node $\pi_{i_0} \wedge \phi_{i_0} = \pi_0 \wedge \phi_0 = \pi \wedge \phi$, and the valuation ρ together with $\gamma_{i_0} = \gamma_0$ obviously satisfies $(\gamma_{i_0}, \rho) \models \pi_{i_0} \wedge \phi_{i_0}$.

The second index is $i_1 = 1$. In this case the path in the graph reduces to the initial edges $E = \{\pi \wedge \phi \xrightarrow{\alpha} \Delta_\alpha(\pi \wedge \phi) | \alpha \in \mathcal{S}\})$ and the path consists of one transition $\gamma_0 \Rightarrow_{\alpha_1} \gamma_1$ for some $\alpha_1 \in \mathcal{S}$ with $(\gamma_0, \rho) \models \pi \wedge \phi$ for some ρ. Then, Lemma 1 ensures $(\gamma_1, \rho) \models \Delta_{\alpha_1}(\pi \wedge \phi)$, and since our graph contains the edge $\pi \wedge \phi \xrightarrow{\alpha} \Delta_{\alpha_1}(\pi \wedge \phi)$ the coverage of paths of length 1 by edges starting in the initial node is settled.

Assume now that we have built the subsequence up to some index $0 < i_m \leq n$. Thus, we have built the sequence $(0 = i_0 < \cdots < i_m)$ and the path $(\pi \wedge \phi =) \varphi_0 \xrightarrow{\alpha_{i_1}} \cdots \xrightarrow{\alpha_{i_m}} \varphi_m$ satisfying the conclusions of the lemma. If $i_m = n$ the conclusion of the lemma holds directly so we can assume $i_m < n$.

We show how to extend the sequence of indices and the path in the graph.

We know that $\varphi_{i_m} \triangleq (\exists Z)\pi_{i_m} \wedge \phi_{i_m}$ is a node in the graph and that $(\gamma_{i_m}, \rho) \models (\exists Z)\pi_{i_m} \wedge \phi_{i_m}$. Consider the configuration γ_{i_m} on the sequence τ. Since $i_m < n$ the configuration γ_{i_m} has a successor on τ i.e., there is $\alpha_{i_m} \in \mathcal{S}$ such that $(\gamma_{i_m}, \rho) \models \mathit{lhs}(\alpha_{i_m})$. By Assumption 4 on the relation $<$, there exists $\alpha \triangleq \pi_l \wedge \phi_l \Rightarrow (\exists Y)\pi_r \wedge \phi_r \in \mathit{min}(<)$ such that $(\gamma_{i_m}, \rho) \models \pi_l \wedge \phi_l$. We distinguish two cases:

- $\alpha \in \mathcal{S}$, and thus, $\mathcal{S} \models \alpha$;
- $\alpha \in \mathcal{G}$, and we obtain $\mathcal{S} \models_{n-1} \alpha$.

Next, using the definition of $\models$ and $\models_{n-1}$, on the (complete) path $\gamma_{i_m} \cdots \gamma_n$ (a nonempty, strict suffix of τ) there exists an index, say, $i_{m+1} \leq n$, and ρ' such that $(\gamma_{i_{m+1}}, \rho') \models \pi_r \wedge \phi_r$. Moreover, $i_{m+1} > i_m$ since $\gamma_{i_m} \in [\![\pi_l \wedge \phi_l]\!]$, which by Assumption 5.2 is disjunct from $[\![\pi_r \wedge \phi_r]\!]$ that contains $\gamma_{i_{m+1}}$.

By Remark 6 we have $\gamma_{i_m} \Rightarrow_\alpha \gamma_{i_{m+1}}$, thus, using Lemma 1 and Def. 8 of derivatives, $(\exists Z)\pi_{i_m} \wedge \phi_{i_m} \Rightarrow_{\alpha^{\mathfrak{s}}} \Delta_\alpha((\exists Z)\pi_{i_m} \wedge \phi_{i_m})$, and $(\gamma_{i_{m+1}}, \rho) \models \Delta_\alpha((\exists Z)\pi_{i_m} \wedge \phi_{i_m})$. We take i_{m+1} to be the next element of the sequence $(0 = i_0 < \cdots < i_m)$, and extend the path $\pi_0 \wedge \phi_0 \xrightarrow{\alpha_{i_1}} \cdots \xrightarrow{\alpha_{i_m}} (\exists Z)\pi_{i_m} \wedge \phi_{i_m}$ with the transition $(\exists Z)\pi_{i_m} \wedge \phi_{i_m} \xrightarrow{\alpha}$

$\varphi_{i_{m+1}}$, where $\varphi_{i_{m+1}} \triangleq \Delta_\alpha((\exists Z)\pi_{i_m} \wedge \phi_{i_m})$ From $(\gamma_{i_{m+1}}, \rho) \models \Delta_\alpha((\exists Z)\pi_{i_m} \wedge \phi_{i_m})$ we obtain that $(\gamma_{i_{m+1}}, \rho) \models \varphi_{i_{m+1}}$.

Thus, we have obtained the next index i_{m+1} in the sequence $(0 = i_0 < \cdots < i_k = n)$ and the next node $\varphi_{i_{m+1}}$ in the path $(\pi \wedge \phi =)\varphi_0 \xrightarrow{\alpha_{i_1}} \cdots \xrightarrow{\alpha_{i_n}} \varphi_n$ in the graph satisfying all the lemma's conclusions. This completes the inductive construction of the elements whose existence is stated by the lemma. □

Theorem 1(soundness). Consider a set of RL formulas $\mathcal{S} \cup \mathcal{G}$. If for all $g \in \mathcal{G}$ the procedure in Figure 7 terminates with *Fail* = *false* then $\mathcal{S} \models \mathcal{G}$.

Proof. We prove by induction on n that $\mathcal{S} \models_n \mathcal{G}$. Let $g \triangleq \pi \wedge \phi \Rightarrow (\exists Y)\pi' \wedge \phi' \in \mathcal{G}$ be arbitrarily chosen, and a complete path $\tau = \gamma_0 \Rightarrow_{\alpha_1} \cdots \Rightarrow_{\alpha_n} \gamma_n$ with $\alpha_1, \ldots, \alpha_n \in \mathcal{S}$, such that $(\gamma_0, \rho) \models \pi \wedge \phi$ for some valuation ρ. In the base case $n = 0$, the theorem is trivial, since this would mean that γ_0 is terminal, which together with $(\gamma_0, \rho) \models \pi \wedge \phi$ contradicts Remark 1.

By induction hypothesis, $\mathcal{S} \models_{n-1} \mathcal{G}$. Thus, we can apply Lemma 10 and obtain $k \geq 0$, a subsequence $(0 = i_0 < \cdots < i_k = n)$, and a path $\pi \wedge \phi = \varphi_0 \xrightarrow{\alpha_{i_1}} \cdots \xrightarrow{\alpha_{i_k}} \varphi_n = (\exists Z)\pi_n \wedge \phi_n$ in the graph constructed by the procedure in Fig. 7 such that $(\gamma_{i_j}, \rho) \models \varphi_{i_j}$ for $j = 0 \ldots k$. We have two cases:

- $match_{\cong}(\pi_n, \pi) = \emptyset$: in this case the procedure ends up at line 9 when building φ_n and since *Fail* = *false* we have $implication(\varphi_n, (\exists Y)\pi' \wedge \phi') =$ *true*, which, by definition of the implication predicate, implies $(\gamma_n, \rho) \models (\exists Y)\pi' \wedge \phi'$, which proves the induction step in this case.

- $match_{\cong}(\pi_n, \pi) \neq \emptyset$: since *Fail* = *false* by construction of the procedure there exists $\alpha \triangleq \varphi_l \Rightarrow \varphi_r \in min(<)$ such that $implication(\varphi_n, \varphi_l) =$ *true*, which implies $(\gamma_n, \rho) \models \varphi_l$. But this is impossible by Remark 1.

The induction step and the proof are now complete. □

Appendix C. Proofs from Section 5

Lemma 3. Consider two patterns φ_1, φ_2 such that $\varphi_1 \sim \varphi_2$. Then, $\varphi_1 \equiv \varphi_2$ if for all patterns φ_1', φ_2' such that $\varphi_1 \Rightarrow^!_{\mathcal{S}^{\mathfrak{s}}} \varphi_1'$ and $\varphi_2 \Rightarrow^!_{\mathcal{S}^{\mathfrak{s}}} \varphi_2'$, $\varphi_1' \sim \varphi_2'$ holds.

Proof. Assume $(\gamma_1, \rho) \models \varphi_1$ and $(\gamma_2, \rho) \models \varphi_2$. Since $\varphi_1 \sim \varphi_2$ we have $\gamma_1 \sim \gamma_2$. In order to prove $\varphi_1 \equiv \varphi_2$ we need to show that if both $last(\gamma_1)$ and $last(\gamma_2)$ exist then $last(\gamma_1) \sim last(\gamma_2)$. Assume then that both $last(\gamma_1)$ and $last(\gamma_2)$ exist. Thus, we have the concrete execution $\gamma_1 \Rightarrow_{\alpha_1} \cdots \Rightarrow_{\alpha_{m-1}} \gamma_m = last(\gamma_1)$ for some $m \geq 1$. By repeatedly applying Lemma 7 we obtain the symbolic execution $\varphi_1 \Rightarrow_{\alpha_1{}^{\mathfrak{s}}}$

$\cdots \Rightarrow_{\alpha_{m-1}{}^{\mathfrak{s}}} \varphi_m$ such that $(\gamma_i, \rho) \models \varphi_i$ for $i = 1, \ldots, m$. Since in particular $(\gamma_m, \rho) \models \varphi_m$ and $\gamma_m = last(\gamma_1)$ is terminal, by setting $\varphi'_1 \triangleq \varphi_m$ we obtain $\varphi_1 \Rightarrow^{!}_{\mathcal{S}^{\mathfrak{s}}} \varphi'_1$. Similarly, we obtain $\varphi_2 \Rightarrow^{!}_{\mathcal{S}^{\mathfrak{s}}} \varphi'_2$. By hypothesis, $\varphi'_1 \sim \varphi'_2$, which implies $last(\gamma_1) \sim last(\gamma_2)$. Using Definition 12, $\gamma_1 \equiv \gamma_2$, and $\varphi_1 \equiv \varphi_2$ follows by Definition 13. □

Lemma 4. If $\mathcal{S} \models \mathcal{G}$ and $\varphi \Rightarrow^{!}_{\mathcal{S}^{\mathfrak{s}}} \varphi'$ then there is φ'' such that $implication(\varphi', \varphi'')$ and $\varphi \Rightarrow^{!}_{\mathcal{S}'^{\mathfrak{s}}} \varphi''$ with $\mathcal{S}' \triangleq \min(<)$.

Proof. From $\varphi \Rightarrow^{!}_{\mathcal{S}^{\mathfrak{s}}} \varphi'$ we obtain γ' and ρ such that $(\gamma', \rho) \models \varphi'$ and γ' is terminal. Using Lemma 2 (repeatedly) we obtain an execution $\gamma_0 \Rightarrow_{\alpha_1} \cdots \Rightarrow_{\alpha_n} \gamma_n = \gamma'$ $(n \geq 0)$ such that $(\gamma_i, \rho) \models \varphi_i$ where $\varphi_0 \triangleq \varphi, \varphi_n \triangleq \varphi'$, and the remaining φ_i are the intermediary patterns in the symbolic execution $\varphi \Rightarrow^{!}_{\mathcal{S}^{\mathfrak{s}}} \varphi'$ (if any).

We now prove $(\diamondsuit)$: there exist $k \geq 0$, a subsequence $(0 = i_0 < \cdots < i_k = n)$, and a symbolic execution $\varphi = \varphi'_0 \Rightarrow_{\alpha'_1{}^{\mathfrak{s}}} \cdots \Rightarrow_{\alpha'_k{}^{\mathfrak{s}}} \varphi'_k$ with $\alpha'_i \in \mathcal{S}'$ such that $(\gamma_{i_j}, \rho) \models \varphi'_j$ for $j = 0 \ldots k$.

The first index is (by definition) $i_0 = 0$. In this case, the valuation ρ together with γ_0 obviously satisfies $(\gamma_0, \rho) \models \varphi'_0 = (\varphi = \varphi_0)$.

Assume now that we have built the subsequence up to some index $0 \leq i_m < n$. Since $i_m < n$ we have $m < k$, and we have the symbolic execution $\varphi = \varphi'_0 \Rightarrow_{\alpha'_1{}^{\mathfrak{s}}} \cdots \Rightarrow_{\alpha'_m{}^{\mathfrak{s}}} \varphi'_m$ with $(\gamma_{i_j}, \rho) \models \varphi'_j$ for $j = 0 \ldots m$. In particular, $(\gamma_{i_m}, \rho) \models \varphi'_m$. Since $i_m < n$ the configuration γ_{i_m} has a successor, i.e., there is $\alpha \in \mathcal{S}$ such that $(\gamma_{i_m}, \rho) \models lhs(\alpha)$. By Assumption 4 on the relation $<$, there exists $\alpha' \in min(<)$ such that $(\gamma_{i_m}, \rho) \models lhs(\alpha')$. We distinguish two cases:

- $\alpha' \in \mathcal{S}$, and thus, $\mathcal{S} \models \alpha'$;
- $\alpha' \in \mathcal{G}$, and thus again $\mathcal{S} \models \alpha'$.

Next, using the definition of $\models$ on the (complete) execution $\gamma_{i_m} \cdots \gamma_n$ there exists an index, say, $i_{m+1} \leq n$ such that $(\gamma_{i_{m+1}}, \rho) \models rhs(\alpha')$. Moreover, $i_{m+1} > i_m$ since $\gamma_{i_m} \in [\![lhs(\alpha')]\!]$, which by Assumption 5 item 2 is disjunct from $[\![rhs(\alpha')]\!]$ that contains $\gamma_{i_{m+1}}$.

By Remark 6 we have $\gamma_{i_m} \Rightarrow_{\alpha'} \gamma_{i_{m+1}}$, thus, using Lemma 1, there exists φ'_{m+1} such that $(\gamma_{i_{m+1}}, \rho) \models \varphi'_{m+1}$. Thus, we have obtained the next index i_{m+1} in the sequences $(0 = i_0 < \cdots < i_k = n)$ and $\varphi = \varphi'_0 \Rightarrow_{\alpha'_1{}^{\mathfrak{s}}} \cdots \Rightarrow_{\alpha'_m{}^{\mathfrak{s}}} \varphi'_m$ with $(\gamma_{i_j}, \rho) \models \varphi'_j$ for $j = 0 \ldots m$. In this way the whole sequence can be built, which completes the proof of $(\diamondsuit)$.

In particular, for $\gamma_{i_k} = \gamma_n = \gamma'$ and $\varphi'_k = \varphi''$ we obtain $(\gamma', \rho) \models \varphi''$. Since γ' is terminal we have $\varphi \Rightarrow^{!}_{\mathcal{S}'^{\mathfrak{s}}} \varphi''$. Since we arbitrarily chose γ' and ρ such that $(\gamma', \rho) \models \varphi'$ and obtained $(\gamma', \rho) \models \varphi''$ we also have $implication(\varphi', \varphi'')$: the lemma is proved. □